University of Fort Hare

Together in Excellence

A Comparison of Open Source Object-Oriented Database Products

A thesis submitted in fulfilment of the requirements of the degree of

Master of Science in Computer Science

At

University of Fort Hare

By

Peter Khayundi

Supervisor: Prof J. Chadwick

December 2009

Abstract

Object oriented databases have been gaining popularity over the years. Their ease of use and the advantages that they offer over relational databases have made them a popular choice amongst database administrators. Their use in previous years was restricted to business and administrative applications, but improvements in technology and the emergence of new, data-intensive applications has led to the increase in the use of object databases.

This study investigates four Open Source object-oriented databases on their ability to carry out the standard database operations of storing, querying, updating and deleting database objects. Each of these databases will be timed in order to measure which is capable of performing a particular function faster than the other.

Acknowledgements

I would like to thank God for giving me the strength and energy to finish this dissertation. I would also like to thank my supervisor, Prof. Jim Chadwick, whose constant encouragement and assistance was vital in me completing this dissertation.

I would like to thank my parents for their support and prayers. I would also like to thank my brothers who have constantly been there for me from the start.

I would like to thank Kudakwashe Chodokufa for being there for me during the good and bad times.

To Sibukele Gumbo, for her assistance and encouragement. Finally, to all my lab mates who have been on this journey together with me. May you all be blessed.

Declaration

I, the undersigned, declare that the work contained in this dissertation is my own original work and has not previously, in its entirety or in part, been submitted at any educational institution for a similar or any other degree awarded.

Signature.....

Date.....

Acronyms

ACID	Atomicity, Consistency, Isolation and Durability
API	Application Programming Interface
CAD	Computer-Aided Design
CASE	Computer-Aided Software Engineering
DBMS	Database Management System
DML	Data Manipulation Language
ER	Entity-Relationship
GUI	Graphical User Interface
GIS	Geographic Information System
IDE	Integrated Development Environment
JSQL	Java Structured Query Language
NQ	Native Queries
OLTP	On-line Transaction Processing
OID	Object Identifier
OODBMS	Object-Oriented Database Management System
ORM	Object-Relational Mapping
OLAP	On-line Analytical Processing
OR	Object-Relational
QBE	Query-By-Example

RAM	Random Access Memory
RDBMS	Relational Database Management System
SODA	Service-Oriented Database Architecture
SQL	Structured Query Language
TPC	Transaction Processing Performance Council
UML	Unified Modelling Language
XML	Extensible Markup Language

Table of Contents

Table of Figures

List of Tables

CHAPTER ONE

INTRODUCTION

1.1 Introduction

This aim of this chapter is to discuss the motivation for studying object-oriented databases and provide direction for the study. It also covers research objectives, the thesis statement, delineations and limitations as well as the significance of the study. An organization of the rest of the dissertation chapters is also covered.

1.2 Background Information

Object-oriented databases have over the past few years been gaining popularity amongst database professionals. These databases offer many advantages over relational databases that can be exploited by those professionals. Some of these advantages include, but are not limited to, the following (Rolland 1998):

1. Circumventing the need for a query language.

2. No impedance mismatch.

3. Eliminating the need for primary keys.

Object-oriented databases also rely on object-oriented programming languages, making them an ideal match for environments such as Java and .Net. Previously, database professionals had to rely on using object oriented programming languages with relational databases, leading to an "object-relational (OR) mismatch". This occurred when relational databases were incompatible with object-oriented programming languages such as Java and .NET (Versant Corp. 2009).

This study was done in order to review a number of Open Source object-oriented databases that are on offer in the market. The results gained from this study may be used to gauge how efficiently the databases were able to perform different database operations.

1.3 Problem Statement

There exist a large number of Open Source object-oriented database products on the market and all offer different levels of functionality as well as performance. Many of these products have not been evaluated against each other to determine which of them provides better performance when it comes to carrying out different database operations.

It is common that a database product that optimizes searching will not necessarily be efficient at storing or modifying data. A user who intends to do a lot of searching may want a database for which searching is optimized. On the other hand, if inserting new data is more frequent, then the user will look for a database for which this operation is optimized. It will obviously be helpful to users to know which database is most efficient when searching, and which is better when it comes to storing new data.

The problem that this dissertation considers is:

- Which Open Source object-oriented databases are better able to perform basic database operations of storing, querying, updating and deleting objects?

In order to answer this question, experiments will be carried out in order to test how well the databases used in this study are capable of performing the above mentioned operations. The results of these experiments will then be used to make reasonable conclusions about the operation of the object databases. No previous studies were found comparing the databases that were used in this study.

There have been previous studies on Open Source databases, such as the one carried out by Mabanza (Mabanza 2006) on XML database, and a comparison of the performance between an object database, Db4o, and an object-relational mapping (ORM) tool, Hibernate (Van Zyl, Kourie & Boake 2006) . These studies served as a guide for the author of this dissertation to carry out a comparison of Open Source object-oriented databases.

1.4 Research Objective

The objective of this project is to experiment on and compare different Open Source object-oriented database products on their ability to perform the basic database operations of storing, querying, updating and deleting database objects.

The experiments that will be carried out on each database will involve calculating the amount of time that each database takes to perform an operation. Each database will be required to perform a specific task that will be timed in order to see how much time is taken to perform the operation.

The results obtained will be tabulated and graphs will be created. These graphs will provide data that will be used to determine which of the databases under experiment is most suitable for performing a particular operation.

1.5 Thesis Statement

The problem statement and research objective sections shown above have led to the thesis of this study:

- Some Open Source object-oriented databases are better than others at performing the standard database operations of storing, querying, updating and deleting database objects.

All databases are required to be able to perform the four basic database operations of storing, querying, updating and deleting. This should also apply to the Open Source object databases to be used in this study.

The databases will be investigated and experimented on to show that they can actually perform the basic database operations. The tests that will be carried out on the object databases are designed to prove whether these databases can perform the operations that are required of

them. The databases will then be compared against each other to see which of them performs a particular operation faster.

1.6 Delineations and Limitations

1.6.1 Delineations

There was some delineation that was made for this dissertation:

1. Only Open Source databases were used. These databases were freely available for download from the Internet. Only databases that came with good documentation and had support in the form of forums and mailing lists were considered. This was to ensure that any difficulties encountered with the databases could be referred to these forums and mailing lists for support.

2. The databases chosen were object-oriented. The aim of this study was to compare the performance of object databases and therefore only these were selected for the study.

3. The objects created within the database were restricted to 100000. This decision to restrict the objects to 100000 was taken as this number was considered adequate for the purposes of comparison.

4. Java was used as the programming language. This was done since this was the language that the author of the dissertation was most comfortable with.

1.6.2 Limitations

There were some limitations experienced during the course of this study. These were:

1. The tests were performed in the Windows environment only. The results obtained may have been different had other operating systems been used to conduct the experiments. At the time of conducting this study, the Windows environment was the most suitable as it was available. Also, the databases chosen for the study were mostly implemented in the Windows environment.

2. Due to time limitations, the number of experiments conducted was restricted. It was felt adequate to carry out a reasonable selection of tests that would measure performance on inserting data, querying, updating and deleting. Future work would involve further testing with more detailed experiments.

3. There was no graphical user interface (GUI) to visualize the contents of the databases. With other commercially available databases such as Microsoft Office® Excel, the contents of the database are visible to the user. The object databases used for this study had no GUI available to view the contents of the databases. This feature would have been useful to enable one to see the actual objects stored within the databases.

1.7 Significance of the Study

This study aims to provide a reasonable comparison of Open Source object-oriented database products that will enable a user to make an informed decision on which of these products would be best suited to carry out a particular database operation. With the many databases available on the market today, users would like to be able to choose which database would suit their particular needs.

This study will compare the performance of different object databases and provide conclusions that will show which of them is most suitable for performing any of the basic database operations of storing, querying, updating and deleting database objects.

1.8 Organization of the Dissertation

Chapter 2 discusses object-oriented database management systems (OODBMSs) and related concepts. The chapter will also define OODBMSs and explain why they are used. It will look at the desired characteristics for an OODBMS. It will also look at the advantages and disadvantages of using OODBMSs as compared to relational databases. There will also be a brief discussion of a related comparison study carried out on an OODBMS and an object-relational mapping (ORM) tool.

Chapter 3 discusses the object-oriented database products under investigation. It will look at the selection criteria that were used when selecting the object databases for the study. It will also look at the factors to be evaluated when testing the databases used for the study, namely functionality, support, performance, and usability. A comparison of the features of each database will also be carried out.

Chapter 4 discusses the evaluation of the databases used in this study. It will look at the evaluation methodology used to test the databases, which includes timing, experimental set-up and a description of the tests carried out on the databases.

Chapter 5 will look at the results obtained from the experiments conducted on the object databases. It will also compare the results in order to establish which database performs a particular operation better in terms of the amount of time taken to complete an operation.

Chapter 6 is the concluding chapter. It summarizes the contents of the dissertation and also proposes future work.

CHAPTER 2

OBJECT-ORIENTED DATABASES AND RELATED CONCEPTS

2.1 Introduction

This chapter will look at object-oriented database management systems (OODBMSs) and related concepts. The chapter will define OODBMSs and explain why they are used. It will look at the desired characteristics for an OODBMS. It will also look at the advantages and disadvantages of using OODBMSs as compared to relational databases. There will also be a brief discussion of a related comparison study carried out on an OODBMS and an object-relational mapping (ORM) tool.

2.2 Object-Oriented Database Management Systems (OODBMSs)

2.2.1 What is an Object-Oriented Database Management System (OODBMS)?

An object-oriented database management system (OODBMS) is a database system that allows objects to be stored and shared between different applications (Rolland 1998). Although object-oriented DBMSs do not have a corresponding theoretical foundation similar to that of relational databases, they all exhibit several common characteristics.

All object-oriented systems rely on the concept of an object. An *object* is a set of related attributes along with the actions associated with the set of attributes. In relational systems, actions are created as part of data manipulation, rather than as part of the data definition. In contrast, in object-oriented systems, the data and actions are encapsulated, which means that an object is defined to contain both the data and its associated actions. Thus an object-oriented DBMS is one in which data and actions that operate on the data are encapsulated into objects (Pratt & Adamski 2005).

2.2.2 Why use Object-Oriented DBMSs?

The first and most important database applications were used in business and administration, mainly for banking applications to maintain information about customers and accounts, and also for applications that stored record-oriented data, such as an inventory system. In the 1980s, new data-intensive applications emerged as a result of hardware innovations. Therefore, traditional DBMSs, based on the relational data model, were inadequate (Chaudhri & Zicari 2001). Examples of some data intensive applications include (Rolland 1998):

1. **Multimedia databases** which require the storage of segments of sound, pictures and text and the ability to associate them together in a consistent manner.

2. **Geographic Information Systems** (GIS) which require the storage of different types of mapping and statistical data which may be subdivided into and collated from overlapping regions.

3. **Design databases** which store data and diagrams involving complex components which may be associated together into further complex components.

Object-oriented DBMSs have been developed in order to meet the requirements imposed by the applications listed above. The object-oriented approach provides the required flexibility because it is not limited to the data types and query languages available in traditional database systems. One of the most important features of OODBMSs is the ability to specify both the structure of complex application objects and the operations to manipulate those structures (Chaudhri & Zicari 2001).

2.3 Object-Oriented Databases: Related Concepts and Characteristics

In order for a system to qualify as an object-oriented database system, there are specific characteristics that it must satisfy. These characteristics serve as a benchmark against which one can measure object-oriented systems. These characteristics are described further below (Atkinson *et al*. 2003):

2.3.1 Complex Objects

An OODBMS must support the creation of complex objects from simple objects such as integers, characters, byte strings of any length, booleans, and floats. According to the Object-Oriented Database Management System Manifesto (Atkinson *et al*. 2003), complex objects are built from simpler objects by applying constructors to them. Examples of complex object constructors include tuples, sets, bags, lists, and arrays. The minimal sets of constructors that a system must have are set, list, and tuple. *Sets* are important because they are a natural way of representing collections from the real world. *Tuples* are important because they are a natural way of representing properties of an entity. *Lists* or *arrays* are important because they capture order, which occurs in the real world, and they also arise in many scientific applications (Atkinson *et al*. 2003).

Supporting complex objects also requires that appropriate operators must be provided for dealing with such objects. This means that operations on a complex object must propagate transitively to all its components. Additional operations on complex objects may be defined by users of the system.

2.3.2 Object Identity

An OODBMS must provide a way to identify objects, i.e. the OODBMS must provide a way to distinguish between one object and another (Atkinson *et al*. 2003). Object identities (OIDs) are usually not directly visible and accessible by database users; they are internally used by the

system. In addition to the object identifier, an object can be characterized by one or more names that are meaningful to the programmer or the end user.

OIDs are used in OODBMSs to identify objects and to support object references through object property values. Objects can thus be interconnected and share components (Chaudhri & Zicari 2001).

2.3.3 Types and Classes

An OODBMS must support types and classes. A *type* summarizes the common features of a set of objects with the same characteristics. It has two parts: the interface and the implementation, with only the interface part being visible to the users of the type. The implementation part of the object is seen only by the designer of the system. The interface consists of a list of operations together with their signatures, i.e. the type of the input parameters and the type of the result (Atkinson *et al*. 2003).

The *type* implementation consists of a data part and an operation part. The data part is used to describe the internal structure of the object's data. The operation part consists of procedures which implement the operation of the interface part.

A *class* provides the implementation for a set of objects of the same type. It also provides primitives for object creation (Chaudhri & Zicari 2001). Classes are used to create and manipulate objects. They contain two aspects (Atkinson *et al*. 2003):

1. An object factory which can be used to create new objects.

2. An object warehouse, which means that there is an extension attached to the class, i.e. the set of objects that are instances of the class.

2.3.4 Encapsulation

An OODBMS must encapsulate data and associated methods together in the database (Atkinson *et al*. 2003). The principle of encapsulation states that an object in the database

encapsulates both the program and data. When storing an object in the database, both the data and the operations to be carried out on the data are stored in the database.

Encapsulation ensures that one can change the implementation of a type without changing any of the programs using that type. Application programs are protected from implementation changes in the lower layers of the system (Atkinson *et al*. 2003).

2.3.5 Inheritance

An OODBMS must support inheritance. For any class, or superclass, one can define a *subclass*, with every occurrence of the subclass being considered to be an occurrence of the superclass. The subclass inherits the structure of the superclass as well as its methods. One can also define additional attributes and methods for the subclass (Atkinson *et al*. 2003).

Inheritance is a powerful reuse mechanism. By using such a mechanism when defining two classes, their common properties, if any, can be identified and factorized in a common superclass (Chaudhri & Zicari 2001).

2.3.6 Overriding and Late Binding

In an object-oriented system, an operation can be defined in a class and inherited by all of its subclasses. This operation can thus have a single name and be used differently on various objects. *Overriding* involves redefining the operation implementation for each class. As a result, a single name denotes different programs and the system manages the selection of the appropriate one during execution.

Binding refers to the association of operations to actual programming code. With late binding, this association does not happen until runtime, i.e. until the user actually invokes an operation. With early binding, operations are associated at compile time. Late binding lets one use the same name for different operations (Atkinson *et al*. 2003).

2.3.7 Extensibility

Any DBMS comes with a set of predefined data types, such as numeric and character. An OODBMS should be extensible, meaning that it is possible to define new data types. Furthermore, the OODBMS should make no distinction between the data types provided by the system and these new data types (Atkinson *et al*. 2003).

Extensibility is one of the key features of object-oriented systems. One can be able to manipulate them in order to suit specific applications. Extensibility can be explained further using the following (InterSystems Corp. 2009: Extensibility):

1. Type definition – One can define new data types that represent application-specific data.

2. Event handling – An application can define a set of methods that are called when a specific action or event occurs. These methods enable the behaviour of an application to be modified without changing the application's core implementation.

3. Subclassing – This involves adapting previously developed components for new uses by creating subclasses of existing classes.

2.3.8 Computational Completeness

One must be able to use functions in the language of the OODBMS to perform various computations. This means that one can express any computable function using the data manipulation language (DML) of the database system (Atkinson *et al*. 2003). Alternatively, an OODBMS should provide an application programming interface (API) to a standard programming language (Casson 1994: Object-oriented databases) so as to enable the programmer to use it.

2.3.9 Persistence

This is the ability of a programmer to have his/her data survive the execution of a process, in order to eventually reuse it in another process. It also refers to the ability to have a program 'remember' its data from one execution to the next (Atkinson *et al*. 2003).

Persistence provides three main benefits (Pratt & Adamski 2005):

1. An application can stop and restart and retrieve the data that it needs to function.

2. An application can work with larger data sets than will fit in memory.
3. An application can share its data with other processes. This has traditionally meant the transfer of data from volatile memory to disk.

2.3.10 Performance

An OODBMS should have sufficient performance capabilities to efficiently manage very large databases (Atkinson *et al*. 2003). Most applications require that databases store thousands of records within them, and it would be useful to have an OODBMS that is capable of handling such large volumes of data.

2.3.11 Concurrency or Concurrent Update Support

An OODBMS must support concurrent update. Concurrent update occurs when multiple users make updates to the same database at the same time (Atkinson *et al*. 2003). Concurrency deals with allowing multiple users to simultaneously access shared entities, such as objects or data records (Ambler 2009: Introduction to concurrency control).

To prevent users from accessing data or records in a database and making undesirable changes, various locking mechanisms are implemented to avoid collisions. A collision occurs when two activities attempt to change entities within a system. Examples of these locking mechanisms include (Ambler 2009: Introduction to concurrency control)**:**

1. Pessimistic locking – This approach ensures that an entity is locked in the database for the entire time that it is in memory. The lock limits or prevents other users from working with the entity in the database.

2. Optimistic locking – This approach considers that collisions may occur infrequently and therefore, instead of trying to prevent them, it detects and resolves then when they occur.

3. Overly-optimistic locking – This approach assumes that collisions will never occur, and it neither tries to avoid nor detect them. It is suitable for a single-user environment.

Deadlocks may occur in database transactions during updates. Records may be locked during updates and may not be accessible to other users (Jenkov [n.d.]: Deadlocks). In order to prevent deadlocks, some techniques can be used (Jenkov [n.d.]: Deadlock Prevention):

1. Lock ordering – Thread locks can be taken in the same order all the time so as to prevent deadlocks. This mechanism is simple yet effective, but it can only be used if one knows about all the locks needed ahead of taking any of the locks.

2. Lock timeout – A timeout may be applied on lock attempts. Any thread trying to obtain a lock will give up if it tries to do so for a long time. If a thread does not succeed in taking all necessary locks within the given timeout, it will back up, free all locks, wait for a random amount of time and retry. The random amount of time will give other threads that may be trying to take the same locks a chance to take all locks. The application will therefore continue running without locking.

3. Deadlock detection – This mechanism is used when lock ordering is not possible and lock timing is not feasible. When a deadlock is detected, one possible action is to release all locks, back up wait a random amount of time and then retry. This option is similar to the lock timeout mechanism. A better option is to assign or determine a priority of the threads so that only one or a few threads back up. The rest of the threads continue taking the locks they need as if no deadlock had occurred.

2.3.12 Recovery Support

An OODBMS must provide recovery services. *Recovery* is the process of returning the database to a state that is known to be correct from a state known to be incorrect.

If the data in a database has been damaged, the simplest approach to recovery involves periodically making a copy of the database, called a *backup*. If a problem occurs, the database is recovered by copying the backup copy over it. The damage is undone by returning the database to the state it was in when the last backup was made (Atkinson *et al*. 2003).

An OODBMS must provide the software tools necessary to implement recovery in the event of an *inconsistent state* arising in the database. Inconsistent states may arise as a result of the following (Hughes 1991):

1. Failure of an updating transaction before it has completed its update but after it has written some changes to the database.

2. A software failure in the operating system or database management system which causes some or all transactions executing at the time of the failure to abort.

3. A power failure which brings all transactions currently active to a halt and loses the contents of main memory.

4. A media failure, such as corruption of a disc.

5. Corruption of the database by a faulty transaction, i.e. a transaction with faulty logic which writes incorrect or inconsistent data to the database.

Many modern DBMSs provide a variety of facilities for protecting against inconsistent states, or for resolving inconsistencies when they arise. Some of these facilities are discussed below (Hughes 1991):

1. Back-up Copies and Snap-shots – Back-up copies of a large database can only be taken as frequently as is cost-effective, as this is an expensive and time-consuming operation.

The copy taken must represent a consistent state, so no updating transaction must be in progress at the same time as the copying utility. In a highly volatile environment, i.e. one in which the information in the database is constantly being updated, frequent 'snap-shots' of highly active areas of the database are desirable.

2. The Log File – Many DBMSs maintain a transaction logging file, or journal file, which records a history of every transaction which has updated the database since the last back-up copy was made. Entries for each transaction in the log file may consist of the following:

    i. A unique transaction identifier.

    ii. The address of every object updated, or created, by the transaction together with the old value of this object and its new value.

    iii. Key points in the progress of transactions, such as their start and end times. It is particularly useful if the log file records the point at which a transaction 'commits', i.e. when it has successfully recorded in the log file all its changes to objects in the database.

3. Recovery from Inconsistent State – This may involve either undoing the changes made by transactions, or redoing the updates of committed transactions. Most recovery strategies require the log to record *checkpoints*, which are simple records written to the log indicating a point in time to which the system can return and be consistent.

2.3.13 Query Facility

An OODBMS must provide query facilities. A query facility should satisfy the following three criteria (Atkinson *et al*. 2003):

1. It should be high level, i.e. one should be able to express, in a few words or in a few mouse clicks, non-trivial queries concisely. This implies that it should emphasize the *what* and not the *how.*

2. It should be efficient, i.e. the formulation of queries should lend itself to some form of query optimization.

3. It should be application independent, i.e. it should work on any possible database.

The characteristics described above are the ideal requirement for a system to be referred to as an OODBMS. The OODBMSs to be used in this study will be required to satisfy all the above requirements in order to be referred to as object-oriented databases. Each of them will be tested exhaustively so as to classify them as OODBMSs.

2.4 OODBMS Advantages and Disadvantages

OODBMSs are often compared to relational database management systems (RDBMS) when it comes to functionality and features. Many of the features that were only available in RDBMSs are now being adapted to OODBMSs. This section will look at the advantages and disadvantages of using an OODBMS as compared to an RDBMS (Obasanjo 2001).

2.4.1 Advantages

1. Composite Objects and Relationships – Objects in an OODBMS can store an arbitrary number of types as well as other objects. Therefore, it is possible to have a large class which holds many medium sized classes which themselves hold many smaller classes. In an RDBMS this has to be done by either having one large table with many null fields or with a number of smaller, normalized tables which are linked through foreign keys. An object is also a better model of the real world entity than the relational tuples when dealing with complex objects.

2. Class Hierarchy – Data in the real world usually has hierarchical characteristics. It is easier to describe such data in an OODBMS than in an RDBMS with the use of superclasses and subclasses.

3. Circumventing the Need for a Query Language – A query language is not necessary for accessing data from an OODBMS, unlike an RDBMS. Interaction with the database is done by transparently accessing objects from the database. However, it is still possible to use queries in an OODBMS.

4. No Impedance Mismatch – When using an object-oriented programming language with an RDBMS, a significant amount of time is usually spent mapping tables to objects and back. This is completely avoided when using an OODBMS.

5. No Primary Keys – In an RDBMS, tuples must be uniquely identified by their values and no two tuples can have the same primary key values in order to avoid error conditions. In an OODBMS, the unique identification of objects is done behind the scenes using object identifiers (OIDs) and is completely invisible to the user. Therefore, there is no limitation to the values that can be stored in an object.

6. One Data Model – A data model typically should model entities and their relationships, constraints and operations that change the states of the data on the system. An RDBMS is not able to model the dynamic operations or rules that change the state of the data in the system as this is beyond the scope of the database. Therefore, applications that use RDBMSs usually have an Entity-Relationship (ER) diagram to model the static parts of the system and a separate model for the operations and behaviours of entities in the application.

   In an OODBMS, there is no separation between the database model and the application model because the entities are just other objects in the system. An entire application can therefore be modeled comprehensively in one Unified Modeling Language (UML) diagram.

2.4.2 Disadvantages

1. Schema Changes – Modifying the database schema in an RDBMS by either creating, updating or deleting tables is typically independent of the actual application. In an OODBMS, modifying the schema in any way means that changes have to be made to the other classes in the application that interact with instances of that class. Consequently, all schema changes in an OODBMS will involve a system-wide recompile. Also, updating all the instance objects within the database can take a long time depending on the size of the database.

2. Language Dependence – An OODBMS is usually tied to a specific language through a specific Application Programming Interface (API). This means that data in an OODBMS is only accessible from the specific language using a specific API, which is not the case with an RDBMS.

3. Lack of Ad-Hoc Queries – In an RDBMS, the relational nature of the data allows one to construct ad-hoc queries where new tables are created from joining existing tables then these new tables are queried. With OODBMSs, it is not possible to duplicate the semantics of 'joining' two classes, thus losing flexibility. Therefore, the queries that can be performed on the data in an OODBMS are highly dependent on the design of the system.

2.5 Benchmarking

A benchmark is a standard by which something can be measured or judged (Seng 1998: under heading Introduction). A database benchmark may be defined as a standard set of executable instructions used to measure and compare the relative and quantitative performance of two or more database systems through the execution of controlled experiments. Benchmarking can therefore be described as a process of evaluating different database software systems on the same or different hardware platforms.

Benchmark data such as throughput, jobs per time unit, and the inverse measure, as well as independent measures such as price performance ratio, equivalent database size, and Web interactions per second will serve to predict and profile the system performance. (Gray 1993).

Database benchmarks comprise test databases and test workloads, which can be synthetic or empirical. Synthetic benchmarks emulate typical applications which can be found in a pre-determined problem domain and create a corresponding synthetic workload. Empirical benchmarks utilize real data and tests and they re-invent the actual database applications. (Seng, Yao & Hevner 2003).

Benchmarks depend on the nature of the tested task (Versant Corp. 2009). Each experiment in a benchmark is made up of two kinds of variables (Seng 1998: under heading Introduction):

1. Experimental factors – Independent variables which will affect the performance of database systems.

2. Performance metrics – Dependent variables which represent quantitative measurements collected from the benchmarking process.

The results produced depend on the following (Seng 1998: under heading Introduction):

1. Workload, which is the amount of work assigned to or performed by a database system in a given period of time.

2. Specific application requirements.

3. System design and implementation.

Examples of some benchmarks include:

1. The HyperModel benchmark – This is an early OODBMS benchmark designed to test hypertext and hyperlink applications (Seng 1998: HyperModel benchmark). It may also be used to test the performance of object-oriented DBMSs for engineering applications

(Berre, Anderson & Mallison 1999). This benchmark tests two of the most important features of OODBMSs: complex object representation and complex object implementation.

2. The 001 benchmark – This benchmark models the common requirements of engineering applications. It defines the common workload characteristic of computer-aided software engineering (CASE) and computer-aided design (CAD) applications. This benchmark assesses the performance of OODBMS, RDBMS, network database systems and hierarchical database systems. The database size can range from 4MB to 400MB (Seng 1998: 001 benchmark).

3. The 007 benchmark – This is an extension of the 001 benchmark where the benchmark database and test sets are expanded to include more complex objects and operations. Some of the performance characteristics that the 007 benchmark tests include the speed of many different kinds of pointer transversals over cached data, the efficiency of many different kinds of updates, and the performance of the query processor on several different types of queries (Carey, DeWitt & Naughton 1993).

4. PolePosition – This is a benchmark test suite to compare database engines and object-relational mapping technology (Sourceforge 2009). Some of the databases that were tested using this benchmark include db4o, Hibernate, MySql, Mckoi, JavaDB, HSQLDB and SQLite. The results of these tests can be found at (Sourceforge 2009).

5. Wisconsin benchmark – This benchmark was developed for relational database systems and machines. It was designed with two objectives in mind: firstly, the queries in the benchmark should test the performance of the major components of the relational database system. Secondly, the semantics and statistics of the underlying relations should be well understood to enable queries to be added easily (DeWitt 1993). A more detailed discussion of the Wisconsin benchmark is found in chapter 4.

6. The TPC suite of benchmarks – The Transaction Processing Performance Council (TPC) was founded to define transaction processing and database benchmarks and to disseminate objective, verifiable TPC performance data to the industry. A transaction could be referred to as a set of operations including disk read/writes, operating system calls, or any form of data transfer from one subsystem to another. A typical transaction, as defined by the TPC, would include the updating to a database system for such things as inventory control (goods), airline reservations (services), or banking (money).

   The TPC benchmarks include those for on-line transaction processing (OLTP) benchmarks e.g. TPC-A (TPC 1992), TPC-B (TPC 1992), TPC-C (TPC 1995), on-line analytical processing (OLAP) benchmarks e.g. TPC-D (TPC 1998), and an electronic commerce benchmark e.g. TPC-W (TPC 2001). The TPC-A and TPC-B benchmarks are already obsolete and have been replaced by the TPC-C benchmark.

The benchmarks discussed above are just a few of the many that are available on the market. In a study carried out by Seng (Seng 1998) to compare the HyperModel, 001 and 007 benchmarks, it was found that the 007 benchmark ranked high as a more comprehensive and complete benchmark method. The 001 benchmark does not support complex object definition and semantic transversals. The HyperModel benchmark is difficult to implement, though it gives more than 17 tests to measure system performance.

The aim of this study was not to develop a benchmark for object-oriented databases, but to adapt some features from different benchmarks to suit the experiments being conducted. One of the features used from benchmarking methods was conducting controlled experiments on different databases. Experiments were created to test the databases in order to arrive at the results that are reported on in chapter 5. The experiments were structured to test how different Open Source object-oriented databases create, query, update and delete objects.

2.6 Related Comparison Studies

Many studies have been carried out on various object databases, object-relational mapping (ORM) tools and relational databases. ORM 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. They are used to eliminate the impedance or object-relational (OR) mismatch (Van Zyl *et al*. 2006)**.** A comparison of their performance and that of object-oriented databases in carrying out various basic database tasks has also been investigated. The use of ORM tools and relational databases is beyond the scope of this study, but a look at performance comparisons when using OODBMSs, ORM tools and relational databases may be useful in providing some insight into the performance comparisons that will be carried out on the object databases for this study.

Benchmarks were used to measure the performance of an OODBMS compared to that of an ORM tool. The benchmark used by (Van Zyl *et al*. 2006) was the 007 benchmark, which was designed to investigate various features of performance. The OODBMS used was db4o, and the ORM tool used was Hibernate.

The study established that db4o, a popular Open Source object database, performed various operations considerably faster than Hibernate, which is also a popular ORM tool. Some of these operations included creation, transversal times, which are the times taken to navigate around the entire object model, queries and modifications, such as inserting and deleting of objects.

A study was also carried out on Open Source Native XML database products by Mabanza (Mabanza 2006). This study investigated and compared the performance of native XML databases in carrying out the four main database functions of storing, reading or selecting, updating and deleting. The databases were subjected to a series of tests that calculated the time taken to perform the database functions. These results were then presented in graphs to show which database performed a specific function faster.

The methods used in testing the performance of the OODBMSs to be used in this study are similar to those used by Mabanza (Mabanza 2006) to test native XML databases.

2.7 Conclusion

This chapter defined OODBMSs and explained their uses as compared to relational databases. It also gave reasons why object-oriented databases are better suited for some applications not supported by relational databases. Some of these applications are multimedia databases, Geographical Information Systems (GIS) and design databases. Some object-oriented concepts and characteristics were also covered. These characteristics were considered essential for a database to have in order to be considered as being an object database.

The advantages of using OODBMSs over relational databases may be desirable when using some applications. It would therefore be of interest to investigate how OODBMSs perform when doing various basic database operations.

The object databases to be used in this study were compared on their ability to carry out the four basic database operations of storing, updating, performing queries, and deleting. Different conclusions were made about the times taken to perform these operations so as to determine which database was best suited for a particular operation. The databases to be used in this study are discussed further in the next chapter.

CHAPTER 3

OPEN SOURCE OBJECT-ORIENTED DATABASE PRODUCTS

3.1 Introduction

The previous chapter covered various OODBMS concepts and characteristics. It looked at the advantages and disadvantages of using OODBMSs as compared to relational databases. It also looked at related comparison studies that were done using different databases.

This chapter will look at the Open Source object-oriented database products that were chosen for this study. It will look at the selection criteria that were used when selecting the object databases. It will also look at the factors to be evaluated when testing the databases used. A comparison of the features of each database will also be carried out.

3.2 Selection Criteria

In order to evaluate the object databases for this study, some criteria were applied in selecting these databases. These criteria ensured that only a particular type of database was selected that was suitable for this study. The criteria are listed below (Khayundi 2008):

1. The database products had to be Open Source and written in Java. The reason for this was because Open Source products were easily available for download from the Internet. The use of Java was preferred as this was the programming language that the author was familiar with.

2. The database products had to have good documentation and sample code. This was to ensure that code examples were available and any difficulties could be addressed by referring to the documentation.

3. The database products had forums on their websites. The availability of forums was to provide support in cases where the author had difficulties when using the databases.

The following database products were reviewed and considered for selection in this study:

1. **Db4o** – This is an Open Source object-oriented database by db4objects, Inc (Versant Corp. 2009).

2. **JODB** – This is an Open Source object-oriented database developed by Mobixess Inc. (Mobixess 2009).

3. **MyOODB** – This is an Open Source object-oriented database developed by Thomas Hazel (MyOoDB 2009).

4. **Neodatis** – This is an Open Source object-oriented database developed by Olivier Smadja, Cristi Ursachi and Marcelo Mayworm (Neodatis 2009).

5. **Ozone** – This is an Open Source object-oriented database developed by ozone-db.org (Ozone 2009).

6. **Perst** – This Open Source object-oriented database is by McObject LLC (McObject 2009).

7. **Prevayler** – This is an Open Source object-oriented database developed by Klaus Wuesterfeld (Codehaus 2009).

Only four of the databases listed above will be discussed in greater detail in a later section of this chapter. Three databases were not selected because they did not meet the criteria for use in this study. The reasons for each of these databases not being selected are given below:

1. **JODB** – This database offers a number of useful features, which include native queries, transaction rollbacks, data backup functionality and indexing to maximize query performance. In spite of all these useful features, JODB does not provide any documentation with the download. In addition to this, the forum on the website was inactive at the time of conducting this research, which prevented the author from getting any assistance on using the database.

2. **MyOODB** – This database is lacking in any form of documentation in its download folder, or support on its website. It has therefore failed to meet the criteria for use in this study.

3. **Ozone** – This database does not provide any documentation within its download folder. The availability of documentation is important in order to enable one to use the database effectively.

3.3 Factors To Be Evaluated

When testing any product, various factors can be used to determine the suitability of a particular product for a specific purpose. By evaluating a product, we can determine the benefits, drawbacks and risks that may be involved in using that product (Mabanza 2006).

For this study, a number of factors were considered when evaluating the products under investigation. These factors will be discussed below.

3.3.1 Functionality

Functionality may be defined as any aspect of what a product can do for a user (Techtarget 2009). Different users may require different functionality from a particular product. Evaluation of a product enables users to gauge exactly what they will get from the product according to their needs and requirements.

The functionality of the products for this study was investigated in order to give users a clear picture of what each product could do. When dealing with object databases, four basic operations were considered: inserting or storing, querying, updating and deleting (Mabanza 2006) of objects. The products chosen for this study were evaluated on their ability to perform these basic functions. Users could then make an informed decision on which database to use according to its strengths and weaknesses.

3.3.2 Support

For the purposes of this study, support will refer to the personal assistance that vendors provide to end-users concerning their particular product (Techtarget 2009).

The level of support offered by a vendor for their product will determine the level of interest from users. Good vendor support will ensure that a product receives good reviews from users, hence increasing its use.

Various forms of support are available for users. These include documentation, forums and mailing lists (Mabanza 2006).The object databases selected for this study all provided some form of support, and each will be discussed further when looking at the individual object database.

3.3.3 Performance

Performance may be measured in terms of two things: efficient resource usage and user perception (Apple 2009). Users may experience problems with a particular application or product for one reason or another. These problems may be brought about due to different aspects of performance. These aspects include (Apple 2009):

1. Computation Performance – This is concerned with the actual operation of the system being used. It includes characteristics such as the number of instructions executed, the overhead experienced while performing an operation, or what method or algorithm to use when carrying out a particular operation.

2. RAM Footprint – The amount of memory required to run an application is an important consideration when developing an application. Applications that consume RAM resources may affect performance and be undesirable to users.

3. Startup Time – Users may be discouraged from using an application which takes a long time to start up. Some applications may run slower when first started but improve on speed as they run for a while. Though this may be adequate for a server-side application, it may not be suitable for a client-side application. The user must be considered when developing an application with regard to the start-up time.

4. Scalability – This refers to how a system or application performs under heavy loads (Wulf 2001). With object databases, one needs to consider how a database will perform as more objects are added to it. For example, whereas a database may be able to handle the creation of 500 objects, the same database may not be able to cope with an instruction to create 50000 objects. Therefore, an application must be designed with scalability in mind to accommodate users' needs.

5. Perceived Performance – This refers to how the application "feels" to the user (Wulf 2001). Users may not be interested in how fast an application is in terms of processing speed. What may be important to users is their experience when using an application. For users, responsiveness is usually a more important factor than speed (Apple 2009).

3.3.4 Usability

Usability is the extent to which a product can be used by specific users to achieve specified goals with effectiveness, efficiency and satisfaction (Spencer 2004). Users would prefer applications which are easier to use and understand as this would enable them to complete their tasks easily and efficiently.

The attributes discussed above were used to evaluate each of the chosen databases for this study to determine which was suited for a specific purpose. A desirable result would be for each database to meet the requirements presented by the above factors. The databases to be used for this study are described further in the following section.

3.3.5 Market Share

This refers to how popular or widely used a particular product is. For any product, its popularity may not necessarily reflect its superiority over other products. Market share is best viewed together with other factors in order to determine how good a product actually is.

3.3.6 Maintenance

Many products require constant maintenance in order to remain current with users' needs. The maintenance provided by the vendors of products should be adequate enough to enable users to have the most recent and current versions of these products.

Maintenance of Open Source products is usually a self-driven process, with experienced developers of these products providing some form of support to less experienced users. Product websites mailing lists and forums are ideal places to find assistance on maintenance of these Open Source products. The more developed a product's site or mailing list, the more support one will be able to find in terms of maintenance.

3.3.7 Scalability

This refers to the ability of a product to handle a large increase in users, workload or in the case of databases, transactions (Wheeler 2009). Open Source products are required to be scalable so that they suit the users' needs.

3.3.8 Flexibility

Flexibility is a measure of how well a product can be used to handle unusual circumstances that is was not originally designed for (Wheeler 2009). With Open Source products, any user can customize the product to meet their needs, as the developer or user has access to the product's source code.

3.4 Indexing

Indexes are used to speed up the retrieval of records in response to certain search conditions (Elmasri & Navethe 1994). They are crucial in speeding up data access operations such as searching and sorting (Korth & Silberschatz 1986).

There are various techniques that can be used for indexing within an object database. Each technique must be evaluated based on (Korth & Silberschatz 1986):

1. Access method – This is the time taken to find a particular data item within the database.

2. Insertion time – This is the time taken to insert a new item. This time will include the time taken to find the correct place to insert the new data item as well as the time it takes to update the index structure.

3. Deletion time – This is the time taken to delete a data item. This will include the time taken to find the item to be deleted as well as the time taken to update the index structure.

4. Space overhead – This is the additional space occupied by an index structure. If the amount of additional space is moderate, then space can be provided in order to achieve improved performance.

The four databases used for this study all supported some form of indexing. These are listed below:

1. Db4o – This database used an instruction in the code to add an index on a particular field of a class. The code segment is shown in Figure 3.1 below:

```
Db4o.configure( ).objectClass(ClassName.class).objectField("field").indexed(true);
```

Figure 3.1: Sample code for indexing in Db4o

In the code segment above, *ClassName* is used to represent the name of the class and *field* is used to represent the actual field that one would want an index created on.

2. Neodatis – With this database, indexing can be declared on various fields of a class. The code segment in Figure 3.2 below illustrates indexing in Neodatis:

```
Odb.getClassRepresentation(ClassName.class).addUniqueIndexOn("className-
index", fieldname, true);
```

Figure 3.2: Sample code for indexing in Neodatis

In the code above, *ClassName* represents the name of the class and *fieldname* represents the actual object that the index is applied to.

3. Perst – With this database, indices were created in order to allow objects to be added into the database. The code segment is shown in Figure 3.3 below:

```
Class Indices( ) {
        Index uniqueIndex;
        Index nonUniqueindex;

}

root = new Indices( );

root.uniqueIndex = db.createIndex(int.class, true);

root.nonUniqueIndex = db.createIndex(int.class, false);
```

Figure 3.3: Sample code for creating indices in Perst

In the code above two types of indices are created, one being unique and denoted as true and the other being non unique and denoted as false.

A key was then added to each of the indices as shown below in Figure 3.4:

```
root.uniqueIndex.put(new Key(intValue1, ClassName));

root.nonUniqueIndex.put(new Key(intValue2, ClassName));
```

Figure 3.4: Sample code for adding items to an index in Perst

In the code above, indices are created on integer values and the objects being stored in the database are of type *ClassName*.

4. Prevayler – This database does not have a specific indexing technique. The creators of the database suggest that one implements an indexing technique that one prefers.

The four databases were compared against each other with and without indexing enabled in order to determine which of the four would perform better. These results are discussed in Chapter Five.

3.5 Open Source Object-Oriented Database Products

3.5.1 Introduction

There are a number of Open Source OODBMSs available on the market. For the purposes of this study, only four were selected.

3.5.2 Db4o

3.5.2.1 Overview and Features

Db4o stands for **D**ata**B**ase for **(4) O**bjects and is an Open Source object-oriented database developed by db4Objects, Inc. It is a very popular object database with users and customers from over 170 different countries. Some of its biggest customers include Boeing, Bosch, Intel, Ricoh and Seagate (Versant 2009).

Db4o was easy to install. A .jar file from the installation folder was added to the CLASSPATH and the database was installed.

When dealing with object-oriented environments and relational databases, one may experience some difficulty when it comes to the transition between the two. This may lead to the programmer or developer having to sacrifice some aspects of her/his application in order to accommodate this "object-relational mismatch". Db4o has been able to offer a solution to this by doing the following (Versant 2009):

1. It eliminates the object-relational mismatch.

2. It is ACID (Atomicity, Consistency, Isolation and Durability) transaction-safe and allows for querying, replication and schema changes during runtime.

3. It allows for up to 55 times faster performance than object-relational mappers, and therefore leaves a smaller footprint.

4. It can run in the same memory process to enhance the reliability of the database, provide for powerful memory and performance tuning and to allow frequent refactoring with one's integrated development environment (IDE).

5. It gives one the ability to modify, optimize and integrate the database engine easily according to one's specific needs.

Db4o stores objects the way they are defined within the application (Versant 2009). Therefore, it is easier to retrieve objects from the database using simple instructions within the application.

When comparing Db4o with the traditional RDBMS, Figure 3.5 below illustrates the difference in how objects are stored within each database (Versant 2009).

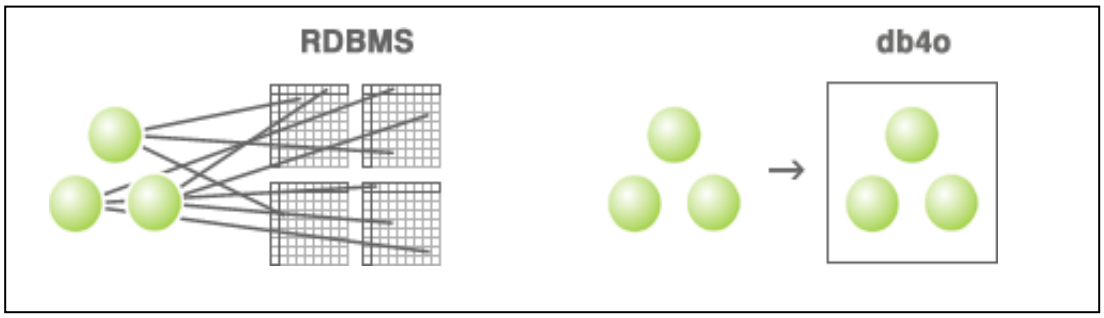

Figure 3.5: Comparison of an RDBMS architecture with that of Db4o

Within an RDBMS, objects are usually mapped into rows and columns to form tables. When programming using an object-oriented programming language, mapping objects to tables can be a long and tedious process (Versant 2009). When creating objects in an OODBMS, what is programmed or created is usually represented the same way in the database. Objects are placed within the database as they are created within the programming code, as shown in Figure 3.4 for Db4o. This eliminates the object-relational mismatch, or impedance mismatch, that occurs when trying to work with RDBMSs while using object-oriented programming languages.

Db4o can be programmed in both Java and .Net (Db4o user guide). For this study, Db4o was programmed using Java.

Db4o supplies three querying systems: Query-By-Example (QBE), Native Queries (NQ) and Service-Oriented Database Architecture (SODA) API. It also supplies methods for updating and deleting objects stored within the database.

Db4o also supports indexing, which can be used to provide maximum querying performance (Db4o guide).

3.5.2.2 Functionality

Db4o supports all the basic functions required of an object database (inserting or storing, querying, updating and deleting). It also supports indexing on various fields of a class. The functionality of Db4o was investigated further in the next chapter, where objects were created, queried, updated and deleted.

3.5.2.3 Support

Db4o provided excellent support in the form of documentation and forums. The forums (Versant 2008) were supported by an active developer community that answered any queries presented to it and was willing to assist with any problems that were encountered during use of the product.

3.5.2.4 Performance

Performance tests were carried out on Db4o and these will be reported on in Chapter Five.

3.5.2.5 Usability

Db4o can be used by users who have an understanding of either the Java or .Net programming environments. At the time of writing this dissertation, it was not known whether the product's developers planned to introduce other programming environments to enable a more diverse array of users to access their product.

The sample code provided for Db4o is easy to understand and follow. A user can be able to easily adapt the code to suit their needs.

3.5.2.6 Market Share

Db4o is a popular product and boasts of many high-profile users such as Boeing, Bosch, Intel, Ricoh and Seagate (Versant 2009). According to Freshmeat (Freshmeat 2009), Db4o has a popularity score of 448.63 and a vitality score of 19.42. Freshmeat calculates the popularity score of a product according to the last 90 days of data collected about this product using the following formula (Freshmeat 2009):

((record hits + URL hits) * (subscriptions + 1))^(1/2)

The vitality score is calculated using the following formula (Freshmeat 2009):

((announcements * age) / (last_announcement))^(1/2)

The number of announcements that the project has made is multiplied by the number of days that it has existed on the Freshmeat database. This is then divided by the days that have passed since the last release. Projects with a high number of announcements that have been around for a long time and have newer releases will earn a high vitality score, with a low vitality score being given to projects that have only been announced once.

There are many users of Db4o as is indicated by the user traffic on its forum, with many users being active.

3.5.2.7 Maintenance

The maintenance of Db4o is an ongoing process. There are improvements offered periodically through email messages and on links on the Db4o website. Feedback on the Db4o forum also provides users with ways on how to make improvements and fix bugs that may be found in the product.

3.5.2.8 Scalability

The scalability of Db4o was tested with the creation of a number of objects and this is reported on in the next chapter.

3.5.2.9 Flexibility

Db4o provides its source code to users and developers for modification. It is therefore flexible and can be tailored to suit the users' needs and requirements.

3.5.3 Neodatis

3.5.3.1 Overview and Features

Neodatis is an Open Source object-oriented database developed by Olivier Smadja and Cristi Ursachi (Neodatis 2009). Some of its users include JConcept, NovaDutra and Tabula (Neodatis 2009).

According to the Neodatis website (Neodatis 2009), these are some of the features of Neodatis:

1. It is simple and easy to learn. The author was able to discover this fact when using Neodatis. It was easy to learn how to create and store objects within the database.

2. It is small in size, with the database runtime size being less than 550KB of data.

3. It is safe and robust. It supports ACID transactions to guarantee the integrity of data in the database. Automatic data recovery during start up ensures that all committed work is applied to the database even in the event of a system failure.
4. It can be used in a multi-threaded environment.

5. It can import and export data to a standard XML format.

6. It can let one persist data with a few lines of code. This ensures that programmers are more productive by allowing them to concentrate more on implementing the business logic than on monitoring the persistence layer.

Neodatis can be run on both Java and .Net platforms. According to the Neodatis website (Neodatis 2009) though, Neodatis currently works on the Java platform and is being ported to the .Net platform.

Neodatis employs four ways of retrieving data. These are:

1. Retrieving all objects of a specific class.

2. CriteriaQuery, which retrieves a subset of objects of a specific class.

3. NativeQuery, which also retrieves a subset of objects of a specific class.

4. Retrieving objects using the object ID (OID).

Future releases will support Query-By-Example, and SQL-like queries (Neodatis 2008).

3.5.3.2 Functionality

Neodatis can perform the storage, querying/retrieval, updating and deleting of objects. It also provides indexing. These were investigated further in the next chapter.

3.5.3.3 Support

Neodatis offered support in the form of a forum on their website (Sourceforge 2009) where uses could post queries about using the database and receive assistance from site administrators and other users.

3.5.3.4 Performance

Performance tests were carried out on Neodatis and will be covered in Chapter Five.

3.5.3.5 Usability

Neodatis is available for users to use with Java and is being ported to .Net and Mono (Neodatis 2008).

The sample code provided for Neodatis was also easy to understand and to adapt. Any user can be able to follow this code and change it in order to suit their requirements.

3.5.3.6 Market Share

The use of Neodatis is not as widespread as with Db4o. Some of its users include JConcept, NovaDutra and Tabula (Neodatis 2009). The product is also popular as is shown by the number of active users on its forum (Sourceforge 2009). Frequent activity on a product's forum page is a good indication of the popularity of the product. Freshmeat does not have any score for this product on its website.

3.5.3.7 Maintenance

Users of Neodatis can seek support for the maintenance of the product by visiting the Neodatis website (Neodatis 2009). Users can get support on the improvements they can make on the database through the website forum. They can report bugs, and request the addition of new features to improve the database.

3.5.3.8 Scalability

The scalability of Neodatis was tested with the creation of a number of objects and this is reported on in the next chapter.

3.5.3.9 Flexibility

The source code for Neodatis is available for modification by users and developers.

3.5.4 Perst

3.5.4.1 Overviews and Features

Perst is an Open Source object-oriented database provided by McObject LLC, which is a company co-founded by Steven T. Graves and Andrei Gorine (McObject 2009). Some companies that use the Perst database include, but are not limited to, CA Wily Technology and Carbon Diem (McObject 2009).

As with Db4o and Neodatis, Perst was easy to install. This was done by adding a .jar file from the installation folder to the CLASSPATH. Some of Perst's features and benefits are listed below (McObject 2009):

1. It is object-oriented. Perst stores data directly in Java objects, which boosts runtime performance. This is due to the elimination of the translation required for storage in relational and object-relational databases.

2. It is compact, with a core that consists of only 5000 lines of code. This ensures that there is a minimal demand on resources due to this small footprint.

3. It is reliable. It supports transactions with the ACID properties, and requires no end-user administration.

4. It supports transparent persistence.

5. It makes its source code available to developers and programmers.

6. It provides extras such as garbage collection, schema evolution, XML import/export, database replication, and support for large databases.

Perst is available as an all-Java embedded database, and can also be implemented in C# for the .Net Framework application (McObject 2009).

For querying, Perst uses JSQL, which is an implementation of SQL which provides a common way of using SQL from within Java to access the database (PC Mag 2009). It also uses indexes to speed up querying in the database (McObject 2009).

3.5.4.2 Functionality

Perst is able to store large volumes of data within the database (McObject 2009). A user can also query, update and delete objects from the database, and also provides indexing. Perst's functionality was investigated further in Chapter Five.

3.5.4.3 Support

Perst provided support on its website (McObject 2009) in the form of a forum. Users could join the forum in order to post questions, opinions or experiences about using the database.

3.5.4.4 Performance

The performance of Perst will be covered in Chapter Five. The database was tested on its capabilities to store, query, update and delete objects.

3.5.4.5 Usability

Perst can be coded in Java and C#. Users with experience of these programming environments will be able to use the database effectively.

The sample code provided for Perst was more complex to work with as compared to the previous two databases mentioned above. However, as user can still be able to quickly adapt the code once they can understand how the Perst database structure works.

3.5.4.6 Market Share

Perst is used in a wide range of markets (McObject 2009), including mobile databases, consumer electronics, telecoms and networking just to mention a few. These varying markets

show that Perst enjoys a large market share and therefore increased popularity. Freshmeat does not have any score for this product on its website.

3.5.4.7 Maintenance

The creator of Perst, McObject LLC, maintains an active forum where users can report bugs in the product. This forum also acts as a sounding board for users and developers who may have questions about the use of the product and where other users can post any improvements that they may have made to the existing product.

3.5.4.8 Scalability

The scalability of Perst was tested with the creation of a number of objects and this is reported on in the next chapter.

3.5.4.9 Flexibility

Perst's source code is available for modification by users and developers.

3.5.5 Prevayler

3.5.5.1 Overview and Features

Prevayler is an object persistent database for Java developed by Klaus Wuestefeld. Some users of Prevayler include Paradigm One and ObjectResourceManager (Prevayler 2006).

In order to install Prevayler, a .jar file was added to the CLASSPATH. This process was similar to that of the other databases used in this study.

The architecture behind Prevayler can be illustrated in Figure 3.6 below (Prevayler 2007), with an explanation to follow thereafter:

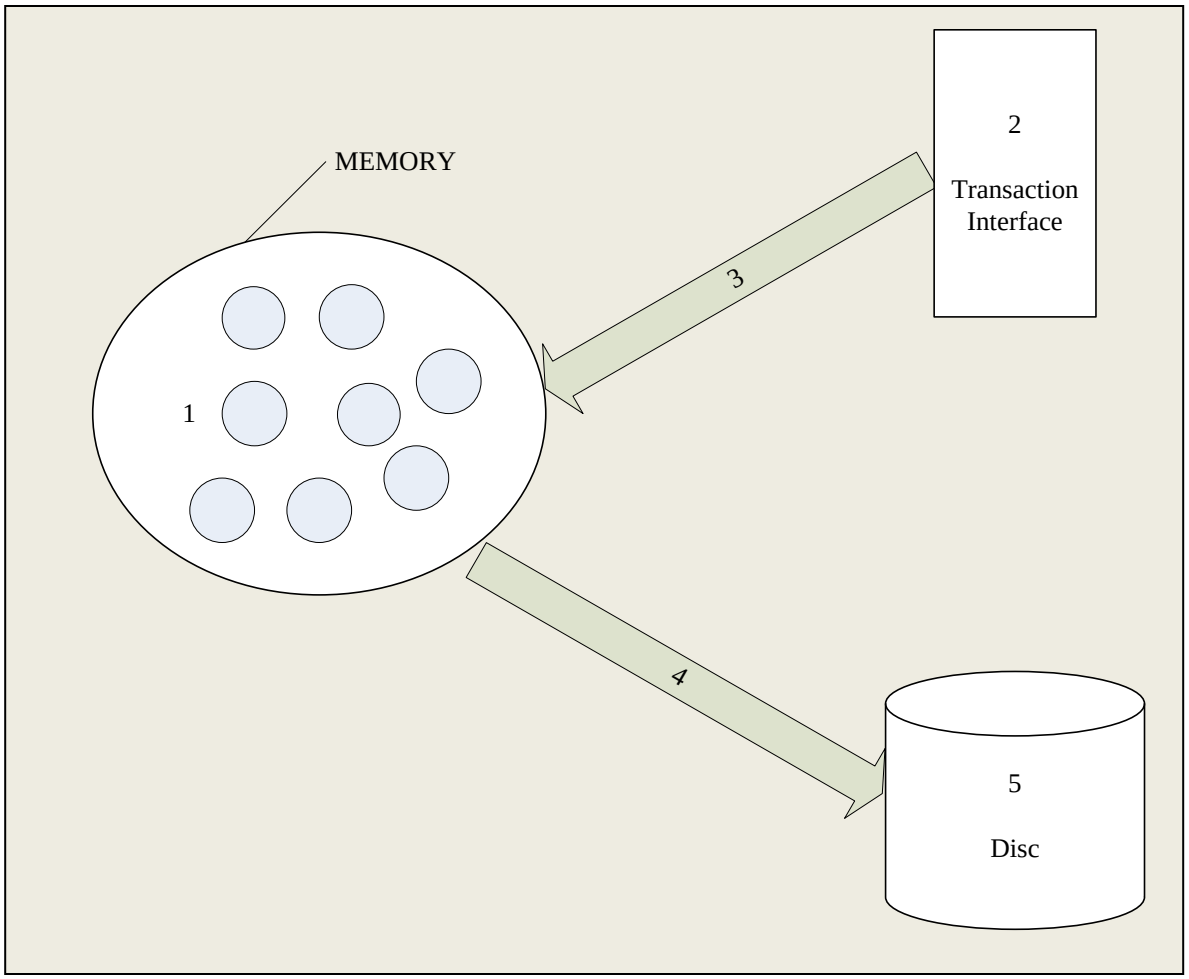

Figure 3.6: Prevayler architecture

The business objects (1) of an application are stored in memory. All modifications to these objects are encapsulated into instances of the transaction interface (2). When Prevayler is asked to execute a transaction on the business objects (3), Prevayler first writes the transaction objects to a journal (4), which are then stored on the disc (5) to prevent data loss in the event of a system crash. Prevayler may also take a snapshot of the business objects and store this on the disc. It can then use this snapshot to automatically recover the business objects when the application is started up again. Some of Prevayler's features are listed below (Prevayler 2007):

1. It is simple, with no separate database server to run.

2. It allows a programmer to program with real objects. One can choose the object models, data structures and algorithms that are suitable for one's application.

3. It is fast, with everything running in memory and the only disc access is streaming transactions to a journal.

4. It makes thread safety easy, with transactions running sequentially. This eliminates multithreading issues such as locking and consistency.

Prevayler is a single-platform database, available only in Java. There is no mention by the developer on whether other platforms will be supported in future releases of Prevayler.

Prevayler does not have a specific way of implementing queries. The developer of the database saw it fit to allow users to adopt any querying mechanism suitable for their purposes (Codehaus 2004).

3.5.5.2 Functionality

According to Prevayler's website (Prevayler 2007), the database is simple and extremely fast. It does not provide any specific form of indexing, and it is left to the user to implement an indexing scheme that one finds suitable for one's needs. The author will test Prevayler and report on whether this is true in Chapter Five.

3.5.5.3 Support

Prevayler provides support in the form of mailing lists (Sourceforge 2009). Users can send mail to an address provided in order to query any of the members on this mailing list for support.

The support provided by Prevayler's mailing lists was very helpful and faster as compared to that offered by the other databases in the form of forums. The response time on the mailing lists was almost immediate and more direct as it targeted the specific queries posted on the mailing list.

3.5.5.4 Performance

The performance of Prevayler will be covered in Chapter 5. The database was also tested on its ability to store, query, update and delete database objects.

3.5.5.5 Usability

Prevayler is available for use only in the Java environment. This limits the use of Prevayler by users who may be familiar with programming environments other than Java, and may prove to work against it when it comes to users choosing a database for their applications.

Prevayler's sample code is very hard to follow and adapt. It involves a very complex object structure consisting of interfaces and abstract classes. A user would find it difficult to adapt their code from that provided by Prevayler.

3.5.5.6 Market Share

Prevayler is not as popular as the other databases used in this study in terms of its use. According to Freshmeat (Freshmeat 2009), Prevayler has a popularity score of 41.38 and a vitality score of 1.41.

3.5.5.7 Maintenance

Prevayler has an active mailing list where users reply to any queries from other users. This mailing list is very helpful, as the author of this dissertation was able to discover when using the product. Other users may also provide assistance with fixing any bugs that may occur during the use of the product.

3.5.5.8 Scalability

The scalability of Prevayler was tested with the creation of a number of objects and this is reported on in the next chapter.

3.5.5.9 Flexibility

Like all Open Source products, Prevayler also provides its source code for modification by users and developers. This ensures that users and developers can modify the code to handle any situation that the product is not initially able to handle.

3.5.6 Summary of Database Features

The databases discussed above can be represented in Table 3.1, with their main features and characteristics listed:

Database	**Developer**	**Features and Characteristics**	**Programming Environments**
Db4o	Db4Objects Inc	Eliminates the object-relational mismatch. Provides indexing.	Java, .Net
Neodatis	Olivier Smadja and Cristi Ursachi	Simple and easy to learn. Can import/export data to a standard XML format. Unique indexing.	Java, .Net, Mono
Perst	Steven T. Graves and Andrei Gorine	Compact. Supports ACID transactions. Provides indexing.	Java, C#
Prevayler	Klaus Wuestefeld	Allows transactions to run sequentially. Runs in memory. Indexing implemented by user.	Java

Table 3.1: Summary of object databases and features

3.6 Conclusion

This chapter covered the databases that were used in this study. It looked at the selection criteria that were used to select the databases for the study. The databases selected were Db4o, Neodatis, Perst and Prevayler. This chapter also looked at factors that were considered important in the evaluation of the databases. These factors were functionality, support, performance and usability. A review of each of these factors was carried out for each database, with performance being discussed further in Chapter Five.

The selection and review of the databases in the previous section provided some insight into their performance under experimentation. All the databases were easy to install and to learn. The author did not have any difficulties when it came to using the databases.

The next chapter will look at the experiments that were carried out on the databases to investigate their performance. These experiments were used to classify the databases according to their speed in carrying out various operations.

CHAPTER 4

EVALUATION OF OPEN SOURCE OBJECT-ORIENTED DATABASE PRODUCTS

4.1 Introduction

In the previous chapter, the object databases used in this study were discussed in detail. Factors considered important in the evaluation of the databases, namely functionality, support, performance and usability, were reviewed. These factors were considered important for a database to have in order for it to satisfy the users' specifications.

This chapter will look at the evaluation of the databases used in this study. It will look at the evaluation methodology used to test the databases, which includes timing, experimental set-up and a description of the tests carried out on the databases.

4.2 Aim

The aim of this study was to provide a way for users to know which database performed a particular function faster, by considering the amount of time a database took to perform a function.

Each database was evaluated according to the amount of time it took to perform any of the basic database functions of creating, querying, updating or deleting objects within the database.

4.3 Research Design

The research design used for this study was a combination of experiments as well as comparative analysis. These methods were chosen because the study involved comparing a number of database products by experimenting on them. Comparative analysis was chosen because a number of products were being tested in order to compare their performance and to draw conclusions from this comparison.

The experiments were conducted in a controlled environment in order to ensure that the results obtained were as accurate as possible. The decision to use experiments to test the databases was taken because it would be possible to predict the results that were to be obtained. Also, the

amount of control over external factors that was provided by experimentation in the lab was desirable for use in this study.

Another advantage of using experiments was that it was possible to control the variables. In the case of this study, some of the variables included, but were not limited to, the number of databases tested and the number of objects stored in each database. In addition, experiments could be replicated, therefore making it possible to authenticate the results produced in the first place (Colorado State University 2009).

Experiments can also be combined with other research methods. Experimental testing was used together with comparative analysis in order to obtain the results used for this study.

The disadvantage of experimentation is that inconsistencies may arise as a result of either human or machine error. Also, though careful attention was taken to ensure that all external factors were considered and controlled, there were still some factors that affected the experiments. Some of these included power outages and lack of access to the lab. In spite of these drawbacks, the experiments were still conducted successfully to produce accurate and reliable results.

The other method used in the research design was comparative analysis. An advantage of this method is that it enables a number of things to be compared against one another. Four database products were compared on their ability to perform the standard database functions of creating, querying, updating and deleting objects. This comparison was used to draw conclusions as to which database was better suited to perform a particular function.

Another disadvantage of comparative analysis is that comparing of a large number of items can prove to be difficult, as one may not be able to fully analyze each product effectively. Though this may be a disadvantage, the choice to analyze four database products for this study was considered sufficient to produce the desired results.

A similar study was carried out by Mabanza (Mabanza 2006) on Open Source XML databases, where a combination of experimentation and comparative analysis was used to compare these database products against each other.

4.4 Evaluation Methodology

For this study, the databases were tested on their ability to perform various functions. Timings were recorded in order to determine which database performed a particular function the fastest. The following sections will describe the research instruments used, the data that were collected and will also include a description of the tests that were conducted.

4.4.1 Research Instruments

The following sections outline the set-up of the experiment, including the hardware and software used when carrying out the tests on the databases, and the installation procedure.

4.4.1.1 Hardware Used

For this study, a single machine was used to install and store all the databases. When the databases were being tested, the machine was new and therefore less likely to experience problems such as hardware or software crashes. The test machine had a Pentium(R) 4, 3.20 GHz processor. It had 2GB of RAM and a hard disk capacity of 160GB. These specifications surpassed all the minimum requirements for installation and operation of all the databases.

The use of a single machine was considered more convenient, as it was desirable to have all the databases in one location to perform the comparisons effectively, and the capacity of the machine was sufficient to cater for all the databases. Furthermore, the experimental results would be distorted by using different processors to test the various products.

4.4.1.2 Software Used

The operating system on the test machine was Microsoft Windows XP Professional N, Version 2002 with Service Pack 2. This operating system came pre-installed with the test machine and was considered sufficient for the purposes of this study.

The databases used were:

1. Db4o version 6.1,
2. Neodatis-odb version 1.8.1,
3. Perst Build 305, and
4. Prevayler version 2.3.

These databases were all Open Source products and were available for download from their respective websites on the Internet. At the time of writing this dissertation, some of the databases already had newer versions available.

Netbeans version 6.0.1 was the integrated development environment (IDE) used to write and debug the code for the databases.

4.4.1.3 Installation

Instructions for the installation of the databases were provided within the installation folders. All the databases were installed by adding a *.jar* file from their installation folders to the CLASSPATH of the test machine. These *.jar* files were also added to the *Libraries* folder in Netbeans.

4.4.2 Data Collected

The data used for this study consisted of timings collected from the database code. Timing data were collected at specific checkpoints in the database code and recorded in milliseconds. The timings recorded were for how long it took for certain database operations to be completed.

In order to obtain the timing data, checkpoints were inserted into the database code. The data were then used to plot graphs that were representative of all the databases and the various operations carried out on them.

4.4.3 The Wisconsin Benchmark

This benchmark was developed to evaluate relational database systems and machines. It became highly successful because it was the first evaluation containing impartial measures of real products (DeWitt 1993). It consists of two parts:

1. A single user benchmark in which a suite of approximately 30 different queries are used to obtain response time measures in standalone mode (DeWitt 1993).

2. A multi-user benchmark in which several queries of varying complexity are used to determine the response time and throughput behaviour under a variety of conditions (Panel discussion 1986).

The creators of this benchmark decided to use synthetically generated relations instead of empirical data from a real database. This was done for the following reasons:

1. Empirical databases are difficult to scale.

2. Values obtained in empirical databases are not flexible enough to permit the systematic benchmarking of the database system.

3. One has to deal with very large amounts of data before it can be safely assumed that the data values are randomly distributed.

The Wisconsin benchmark is composed of three relations, one with 1000 tuples named ONEKTUP, and two others each with 10000 tuples, named TENKTUP1 and TENKTUP2 respectively. The TENKTUP1 relation contains various attributes which include *unique1* and *unique2*. These are uniformly distributed unique random numbers in the range 0 to MAXTUPLES-1, where MAXTUPLES is the cardinality of the relation. The attributes found in each of the relations mentioned above are shown in Table 4.1 below (DeWitt 1993).

Attribute Name	Range of Values	Order	Comment
unique1	0-(MAXTUPLES-1)	random	unique, random order
unique2	0-(MAXTUPLES-1)	sequential	unique, sequential
four	0-3	random	(unique1 mod 4)
twenty	0-19	random	(unique1 mod 20)
onePercent	0-99	random	(unique1 mod 100)
tenPercent	0-9	random	(unique1 mod 10)
twentyPercent	0-4	random	(unique1 mod 5)
fiftyPercent	0-1	random	(unique1 mod 2)
stringu1	-	random	candidate key
stringu2	-	random	candidate key
string4	-	cyclic	

Table 4.1: Attribute Specification of "Scalable" Wisconsin Benchmark Relations

The values of *unique1* are randomly distributed unique values between 0 to MAXTUPLES-1, whereas the values for *unique2* are in sequential order from 0 to MAXTUPLES-1. The values for the *four* and *twenty* attributes are randomly distributed as they are generated by computing the appropriate mod of *unique1*. The other set of attributes, *onePercent*, *tenPercent*, *twentyPercent* and *fiftyPercent*, have been defined so as to simplify the task of scaling selection queries with a certain selectivity factor. For example, the predicate "*twentyPercent = 3*" will always return 20% of the tuples in a relation, regardless of the relation's cardinality.

The string attributes, *stringu1* and *stringu2*, are string analogies of *unique1* and *unique2*. Both *stringu1* and *stringu2* consist of seven significant characters from the alphabet (A-Z) followed by 45 x's. The seven significant characters of *stringu1* (*stringu2*) are obtained by converting the value of *unique1* (*unique2*) to its corresponding character string representation, and padding the result to a length of seven characters with A's.

The last string attribute, *string4*, assumes four values, AAAAxxx…, HHHHxxx…, OOOOxxx…, and VVVVxxx… in a cyclic manner:

AAAAxxx…
HHHHxxx…
OOOOxxx…
VVVVxxx…
AAAAxxx…
HHHHxxx…
OOOOxxx…
VVVVxxx…

The structure of the attributes mentioned above was modified in order to develop the Item class for use in the experiments that were conducted for this study. The way that the attributes were defined made it ideal to use them to model the Item class.

4.4.4 The Item class

In order to evaluate each database, objects must be inserted, queried, modified and deleted. A decision was made as to what kind of objects should be stored in the databases, in order to allow for operations that could be meaningfully timed and would allow for a sensible evaluation.

The approach that was adopted was similar to the one used in the Wisconsin Benchmark for relational databases, as described by DeWitt (DeWitt 1993), where the fields in each row of the database table are structured in such a way as to control the number of rows returned by a specific query.

All objects are of class Item. An Item object has a number of member variables, as depicted in Figure 4.1 below.

.

```
public class Item {
        int num;
        int unique1, unique2;
        int onePercent, tenPercent;
        int twentyPercent, fiftyPercent;
        String stringu1, stringu2, string4;
        // various constructors and other methods
}
```

Figure 4.1: Member variables of the Item class

The variables *unique1* and *unique2* have unique values in a particular database. If the database has *size* objects, then the values of *unique2* are 0, 1, 2, …. (*size* – 1) in the order in which the objects were stored. The variable *unique1* consists of the numbers 0, 1, 2, …. (*size* – 1) in a randomized order. A query of the form (*unique2* < 5) would select the first five items stored in the database, while a query of form (*unique1* < 5) would select five items scattered in various places throughout the database. The query (*unique2* > size – 6) will select the last five items stored in the database.

Variables *onePercent*, *tenPercent*, *twentyPercent* and *fiftyPercent* are defined in the following way: *onePercent* = *unique1 % 100*, *tenPercent* = *unique1 % 10*, *twentyPercent* = *unique1* % 5 and *fiftyPercent* = *unique1* % 2. The query (*fiftyPercent* = 1) will return exactly half of the items in the database. Furthermore, the items selected by this query will be randomly dispersed throughout the database.

The string variables *stringu1*, *stringu2* and *string4* are selected as follows. Firstly, *stringu1* is derived from the unique integer *unique1*, in such a way that each value of the variable *stringu1* occurs exactly once in the database. This is the same for *stringu2*, which is derived from *unique2*. For the variable *string4*, it has exactly four equally likely values, so that each value of *string4* occurs one quarter of the time in the database. The string variables contain about 50 characters, and can be used to test the performance of the database when managing strings of characters.

Finally, the integer variable *num* is only used as a spare variable when updating the database – the value of *num* can be changed without affecting the more significant member variables of the Item class. The complete Item class can be found in Appendix A.

4.4.5 Timing

For each database, the basic operations were performed and timed. The timing was done by calculating how long a database took to perform a particular function. The time (*t1*) recorded before the function was carried out was subtracted from the time (*t2*) after the function was completed. These timings were then used to determine which database performed a particular function faster. Each experiment was repeated ten times, and the mean time, as well as the standard deviation, was calculated.

4.4.6 Description of Tests

The databases were tested in order to evaluate them. These tests were carried out in order to establish the performance of the databases and to show which was capable of performing a specific function faster than the other. These tests are described below.

4.4.6.1 Inserting

Databases were created to store large amounts of data conveniently. A database should be able to handle the storing or insertion of large amounts of data. The databases for this study were required to handle the creation of numbers of objects ranging from 5000 to 100000. The limit of 100000 was chosen as this was considered to be adequate for the purposes of comparison.

A sample of the code to insert objects in db4o is shown in Figure 4.2 below:

```
int limit = 100000;
      for (int size = 5000; size<=limit; size+=5000) {
            long sum = 0;
            long sumSquares = 0;
            long Time = 0;
                for (int m = 0; m<10; m++) {
                        String dbname = ("db4o-no-index-"+size+"-"+m);
                        db = Db4o.openFile(dbname);
                         Item[] items = new MakeItems(size).itemList();
                         long t1 = System.currentTimeMillis();
                                for(int k = 0; k<size; k++){
                                        db.set(items[k]);
                }
        long t2 = System.currentTimeMillis();
        p.println("Size = " + size + "\t" + "m = " + m + "\t" + "time = " + (t2-t1));
        Time = t2-t1;
        sum += Time;
        sumSquares+= Time*Time;
}
        double variance = ((double) (sumSquares - sum*sum/10) /10);
        double SD = Math.sqrt(variance);
        p.println("Average time for size " + size + "\t" + (sum /10));
        p.println("SD = " + SD);
```

Figure 4.2: Sample code for inserting objects in Db4o

In the figure above, *limit* represents the maximum number of objects to be created in the databases, which were 100000. The variable *size* will be used to create databases ranging from 5000 to 100000 in steps of 5000. The variables *sum*, *sumSquares* and *Time* were used to calculate the standard deviation. These objects were placed in databases with corresponding names using the instruction *new MakeItems(size).itemList( )*, which creates an array of *size* items, with values of *unique1*, *unique2* in range *0* to *size-1*. For example, for 5000 objects, these were placed in a database named *db4o-no-index-5000-0*. The instruction *db.set(items[k])* was used to place the objects into the database. The average and standard deviation are calculated and stored in a file on the disc.

The creation of each database was repeated 10 times in order to get the average time for the particular operation. This was done in order to account for variations in the execution time. An average of the timings taken by repeating the operation 10 times was a more accurate

representation of the actual time taken to perform the operation. The standard deviation was also calculated to as to record the spread in time of the creation of the objects due to the repetition.

4.4.6.2 Querying

Every database needs to have a querying facility, where data can be retrieved in some way. The databases chosen for this study all offered such a facility. Each of them was queried in order to compare the response time that would be obtained after a query was performed.

The various databases had different ways of querying. These included:

1. Query-By-Example – This method was available for db4o.

2. Native Queries – This method was provided by both db4o and Neodatis.

3. Service-Oriented Database Architecture (SODA) API – This method was available for db4o.

4. Criteria Query – This method of querying was provided by Neodatis.

5. JSQL – This method of querying was available for Perst.

The databases were queried using specific criteria in order to obtain desired results. The queries used are specified below:

Q1. Querying for *unique1*

The value of *unique1* in the database, as mentioned in section 4.4.4, consists of the numbers 0, 1, 2, …. (*size* - 1) in a randomized order. An example of a query would be to find all items in the database where the value of *unique1* is 300. This is shown in Figure 4.3 below, with sample code from Perst:

```
int limit = 100000;
        for (int size = 5000; size <= limit; size+=5000) {
                long sum = 0;
                long sumSquares = 0;
                int repeats = 1000;
                   for(int m = 0; m < repeats; m++) {
                      long t1 = System.currentTimeMillis();
                      String dbname = "perst-no-index-" + size + "-" + m;
                      Storage db = StorageFactory.getInstance().createStorage();
                      db.open(dbname,Storage.INFINITE_PAGE_POOL);
                      IPersistentList root = (IPersistentList) db.getRoot();
                      Query query = db.createQuery();
                      query.prepare(Item.class, "unique1 = 300");
                      iterator = query.execute(root.iterator());
                      long t2 = System.currentTimeMillis();
                }
        sum += (t2-t1);
        sumSquares+= (t2-t1)*(t2-t1);
        db.close();
}
double variance = ((double) (sumSquares - sum*sum/repeats) /repeats);
double SD = Math.sqrt(variance);
p.println("Searching time for size " + size + "\t" + (((double)sum) /repeats));
 p.println("SD = " + SD);
}
```

Figure 4.3: Sample code for querying for *unique1* in Perst

In the code segment above, the instruction *query.prepare(Item.class, "unique1 = 300");* ensures that the field *unique1* in the Item class will be searched for. The code *iterator = query.execute(root.iterator());* will execute the query and return all objects that match the query. This process is then timed, with the average time and standard deviation being calculated and saved to a file.

Q2. Querying for *unique2*

The code for querying for *unique2* is similar to the one found above, with the only change being in the line *query.prepare(Item.class, "unique2 = 300");* . This code is also timed and the average time and standard deviation calculated and saved to a file.

Q3. Querying for one percent of objects, randomly distributed

This query was used to select one percent of objects from the database. For example, for a database containing 5000 objects, running this query would produce 50 objects. This query was timed for all the databases in order to see which database performed the query faster.

The code for selecting one percent in Db4o is shown in Figure 4.4 below:

```
for (int size = 5000; size <= limit; size+=5000) {
    long sum = 0;
    long sumSquares = 0;
        for (int m = 0; m<10; m++) {
            long t1 = System.currentTimeMillis();
            String dbname = ("db4o-no-index-"+size+"-"+m);
            db = Db4o.openFile(dbname);
            Query query = db.query();
            query.constrain(Item.class);
            query.descend("onePercent").constrain(new Integer(5));
            ObjectSet result = query.execute();
            long t2 = System.currentTimeMillis();
                while(result.hasNext()) {
                        System.out.println(result.next());
                }
            db.close();
            sum += (t2-t1);
            sumSquares+= (t2-t1)*(t2-t1);
        }
        System.out.println("Searched size = " + size + " For onePercent = 5");
        double variance = ((double) (sumSquares - sum*sum/10) /10);
        double SD = Math.sqrt(variance);
        p.println("Searching time for size " + size + "\t" + (((double)sum) /10));
        p.println("SD = " + SD);
}
```

Figure 4.4: Code for selecting one percent in Db4o

The code for selecting one percent in Neodatis is shown in Figure 4.5 below:

```
        for (int size = 5000; size <= limit; size+=5000) {
                long sum = 0;
                long sumSquares = 0;
                int repeats = 10;
        for(int m = 0; m < repeats; m++) {
          long t1 = System.currentTimeMillis();
          String ODB_NAME = ("neodatis-no-index-"+size+"-"+m);
          odb = ODBFactory.open(ODB_NAME);
          IQuery query = new CriteriaQuery(Item.class, Where.equal("onePercent", 5));
          Objects value = odb.getObjects(query);
          long t2 = System.currentTimeMillis();
          odb.close();
          sum += (t2-t1);
          sumSquares+= (t2-t1)*(t2-t1);
        }
        System.out.println("Searched size = " + size + " For onePercent = 5");
        double variance = ((double) (sumSquares - sum*sum/repeats) /repeats);
        double SD = Math.sqrt(variance);
        p.println("Searching time for size " + size + "\t" + (((double)sum) /repeats));
        p.println("SD = " + SD);
    System.out.println("The time taken to select unique1 is "+(((double)sum) /repeats));
        }
```

Figure 4.5: Code for selecting one percent in Neodatis

The code for querying one percent of the objects in Perst is shown in Figure 4.6 below:

```
        for (int size = 5000; size <= limit; size+=5000) {

                long sum = 0;
                long sumSquares = 0;
                 int repeats = 1000;
                        for(int m = 0; m < repeats; m++) {
                                long t1 = System.currentTimeMillis();
                                String dbname = "perst-no-index-" + size + "-" + m;
                        Storage db = StorageFactory.getInstance().createStorage();
                        db.open(dbname,Storage.INFINITE_PAGE_POOL);
                        IPersistentList root = (IPersistentList) db.getRoot();
                        Query query = db.createQuery();
                        query.prepare(Item.class, "onePercent = 5");
                        iterator = query.execute(root.iterator());
                        long t2 = System.currentTimeMillis();
                        sum += (t2-t1);
                        sumSquares+= (t2-t1)*(t2-t1);
                        db.close();
```

```
            }
        System.out.println("Searched size = " + size + " For onePercent = 5");
        double variance = ((double) (sumSquares - sum*sum/repeats) /repeats);
        double SD = Math.sqrt(variance);
        p.println("Searching time for size " + size + "\t" + (((double)sum) /repeats));
        p.println("SD = " + SD);listResult(result);
        }
```

Figure 4.6: Code for selecting one percent in Perst

The code for selecting one percent in Prevayler is shown in Figure 4.7 below:

```
        List list = ((ItemKeeper)prevalentSystem).itemList();
            List result = new ArrayList();
            for(int k = 0; k < list.size(); k++ ) {
               Item item = (Item)list.get(k);
               if(item.onePercent == 5)result.add(item);
            }
            return result;
```

Figure 4.7: Code for selecting one percent in Prevayler

Q4. Querying for ten percent of objects, randomly distributed

This query was used to select ten percent of the objects stored in the database. The code for this query was similar to that for selecting one percent, with the only change being made in the following lines in the respective databases:

- Db4o:

  query.descend(“tenPercent”).constrain(new Integer(4));

- Neodatis:

  IQuery query = new CriteriaQuery(Item.class, Where.equal(“tenPercent”,4));

- Perst:

  query.prepare(Item.class, "tenPercent = 4");

- Prevayler:

  If(item.tenPercent==4) result.add(item);

By modifying the code in the ways shown above, the query was able to select ten percent of the objects stored in the respective database.

The queries above were used to select ten percent of objects stored in each of the databases. A similar modification was made to the code as was done for the previous query to select twenty percent of the objects. The code was then used to select twenty percent of the objects in the database.

Q6. Querying for fifty percent of objects, randomly distributed

This query was executed and it selected fifty percent of the objects that were stored in the database. The code for querying was also changed as in the previous cases to enable it to select fifty percent of the objects stored in the database.

When querying for, say, 50% of the objects in the database, the number of objects returned will clearly increase with the size of the database. This is realistic, in the sense that a particular query is likely to return more results from a large database than from a small one. However, one might also like to compare the timings in the case where the query returns a fixed number of items, independent of the database size.

To test this out, the following two queries were also implemented:

Q7. Querying for the first 1000 objects in the database

This is implemented using the condition that unique2 < 1000.

Q8. Querying for 1000 randomly dispersed objects

This is implemented by requiring that unique1 < 1000.

Q9. Querying strings

The *Item.java* class also had string variables, namely *stringu1*, *stringu2* and *string4*. The values for these variables were obtained by converting the values created for *unique1* and

unique2 into strings. These string variables were then queried in the database and the timings taken to establish how long it took to obtain a result.

This query uses a function *convert( )* that takes any integer as a parameter and converts it to its corresponding string object in the database. It also adds an additional 45 x's. The following queries were used in Perst:

- query.prepare(Item.class, "stringu1 = '"+convert(300)+ x45 +"'");
- query.prepare(Item.class, "stringu2 = '"+convert(300)+ x45 +"'");
- query.prepare(Item.class, "string4 = 'AAAA"+x45+ "xxx'");

The query above for *string4* returns four A's in addition to 45 x's and an additional three x's. The searches based on *stringu1* and *stringu2* return a unique value, and those based on *string4* return one quarter of the objects found in the database.

Q10. Querying for unique fields

The Item class has four fields which are unique, namely *unique1*, *unique2*, *stringu1* and *stringu2*. Experiments were carried out to find out the amount of time it took to search the databases for these fields.

4.4.6.3 Updating

One may want to modify data already stored within the database by either changing it or adding something to its value. Different experiments were conducted in order to measure the time taken to update any particular database. The times were used in order to enable the comparison of the databases when updating a number of objects.

A sample of the code for updating in db4o is shown in Figure 4.8 below:

```
for (int size = 5000; size <= limit; size+=5000) {
    long sum = 0;
    long sumSquares = 0;
    int repeats = 10;
    long t1 = System.currentTimeMillis();
       for (int m = 0; m<repeats; m++) {
           String dbname = ("db4o-no-index-"+size+"-"+m);
           db = Db4o.openFile(dbname);
           Db4o.configure().objectClass(Item.class).cascadeOnUpdate(true);
           Query query = db.query();
           query.constrain(Item.class);
           query.descend("unique1").constrain(new Integer(300));
           ObjectSet result = query.execute();
               for(int k=0; k<result.size(); k++) {
                   Item found = (Item)result.next();
                   found.addNumber(k);
                   db.set(found);
                   db.commit();
                   }
                   db.close();
       }
    long t2 = System.currentTimeMillis();
    sum += (t2-t1);
    sumSquares+= (t2-t1)*(t2-t1);
    System.out.println("Modified size = " + size + "For unique1 = 300");
    double variance = ((double) (sumSquares - sum*sum/repeats) /repeats);
    double SD = Math.sqrt(variance);
    p.println("Modifying time for size " + size + "\t" + (((double)sum) /repeats));
    p.println("SD = " + SD);
}
```

Figure 4.8: Sample code for updating in Db4o

In order to update objects within the database, they must first be queried. This ensured that the update was performed only on the required object or objects. Using the *set*( ) method above without first retrieving the objects would add new objects to the database (Db4o user guide 2003).

Once the objects were queried, a loop was used to go through and modify them by adding a number to the *num* variable in the class Item. This variable is used for no other purpose. The instruction *found.addNumber(k)* added a number *k* to the result obtained from the query. The

changed object was then put back into the database using the *db.set(found)* instruction. The changes to the database were then made using the instruction *db.commit( )*.

Although various databases implemented updating differently, the concept was generally the same. Objects were first retrieved using a query before being updated and returned into the database. The results of the comparison in timings taken to update objects will be discussed in the next chapter.

4.4.6.4 Deleting

The last major function that a database must be able to perform is the deletion of objects. Each of the databases that were chosen was able to delete objects.

The section of code in Figure 4.9 below shows how objects were deleted in db4o:

```
for (int size = 5000; size <= limit; size+=5000) {
    long sum = 0;
    long sumSquares = 0;
    String dbname = ("db4o-no-index"+size+"-"+0);
    db = Db4o.openFile(dbname);
    Db4o.configure().objectClass(Item.class).cascadeOnDelete(true);
    int repeats = 10;
    long t1 = System.currentTimeMillis();
   for(int m = 0; m<repeats; m++) {
        Query query = db.query();
        query.constrain(Item.class);
        query.descend("unique1").constrain(new Integer(300));
        ObjectSet result = query.execute();
        for(int k=0; k<result.size(); k++) {
            Item found = (Item)result.next();
            db.delete(found);
            db.commit();
            System.out.println("Deleted: "+found);
            }
    long t2 = System.currentTimeMillis();
    sum += (t2-t1);
    sumSquares+= (t2-t1)*(t2-t1);
    System.out.println("Deleted size = " + size + "For unique1 = 300");
```

```
        double variance = ((double) (sumSquares - sum*sum/repeats) /repeats);
        double SD = Math.sqrt(variance);
        p.println("Deleting time for size " + size + "\t" + (((double)sum) /repeats));
        p.println("SD = " + SD);
    }
```

Figure 4.9: Sample code for deleting objects in Db4o

As with the updating of objects, one must first query the objects from the database so as to delete them. After the query was executed, a loop was used to go through all the results. Each result was then deleted from the database using the instruction *db.delete(found)*. An instruction was used to check on whether these objects were successfully deleted from the database.

As with updating, the way that the databases deleted objects was similar, with the difference only being in the way the code was written. A query was used to obtain the objects and they were then deleted from the database. The timings were taken to establish which database deleted objects faster. These results will be discussed in the next chapter.

4.5 Conclusion

This chapter covered the methodology used for testing the databases chosen for this study. It covered the research design and evaluation methodology used. The software and hardware environments were discussed and the tests that were performed on the databases were described in detail. These tests were used to compare the performance of the databases in carrying out various functions. The results of these tests will be discussed in the next chapter.

CHAPTER 5

RESULTS AND DISCUSSION OF OUTCOMES

5.1 Introduction

In the previous chapter, the methodology used for this study was discussed. This methodology included the research design and evaluation methodology that was used in this study. Also, the tests that were performed on each database were explained further. The hardware and software environments were also discussed.

This chapter will look at the results obtained from the experiments conducted on the object databases. It will also compare the results in order to establish which database performs a particular function better in terms of the amount of time taken to complete an operation.

5.2 Results Obtained From Experiments

The purpose of the experiments was to perform timings of various databases as they performed the basic database functions of storing, querying, updating and deleting. The timings were a measure of the response time for particular databases for particular operations.

The number of objects that were added to the database ranged from 5000 to 100000. This operation of storing objects was timed and graphs were created to compare the timings from each database. Objects were also queried, updated and deleted and these results will be discussed further in the coming sections.

The results obtained are represented graphically in the following sections. The source code for all the operations, namely storing, querying, updating and deleting, and tables containing the timing results can be found in Appendices A and B respectively.

5.2.1 Storing

The experiments for storing objects within the database comprised of code written to store between 5000 and 100000 objects in the databases. Two sets of experiments were performed,

one with indexing enabled, and the other without. Three databases provided indexing, namely Db4o, Neodatis and Perst. The creators of Prevayler suggest that any form of indexing can be applied according to the users' requirements. In view of this, it was felt that implementing the author's own indexing mechanism would not represent a fair comparison.

For Neodatis, the indexing mechanism only allowed for the creation of unique indexes on any field. The Item class has four fields which are unique, namely *unique1*, *unique2*, *stringu1* and *stringu2*. A different comparison graph was created for the databases with indexing implemented on four fields.

These operations were timed for each database to determine which one performed the operations faster. The timings are shown in milliseconds. The timing differences in each of the graphs with the objects raging from 5000 to 100000 are sometimes small and not noticeable. These differences would be significantly larger with objects in the millions or even billions being added to any particular database. Figure 5.1 below shows a comparison graph for the timings of the four databases when creating without indexing applied. A discussion follows thereafter. The standard deviations (SD) obtained when calculating the times for Perst are very low, as compared to those for Prevayler, which are very high. These standard deviation values imply that the results of the Prevayler experiments were very widely spread as compared to those for the other databases. The times taken by all the databases can be found in a table B.1 in Appendix B.

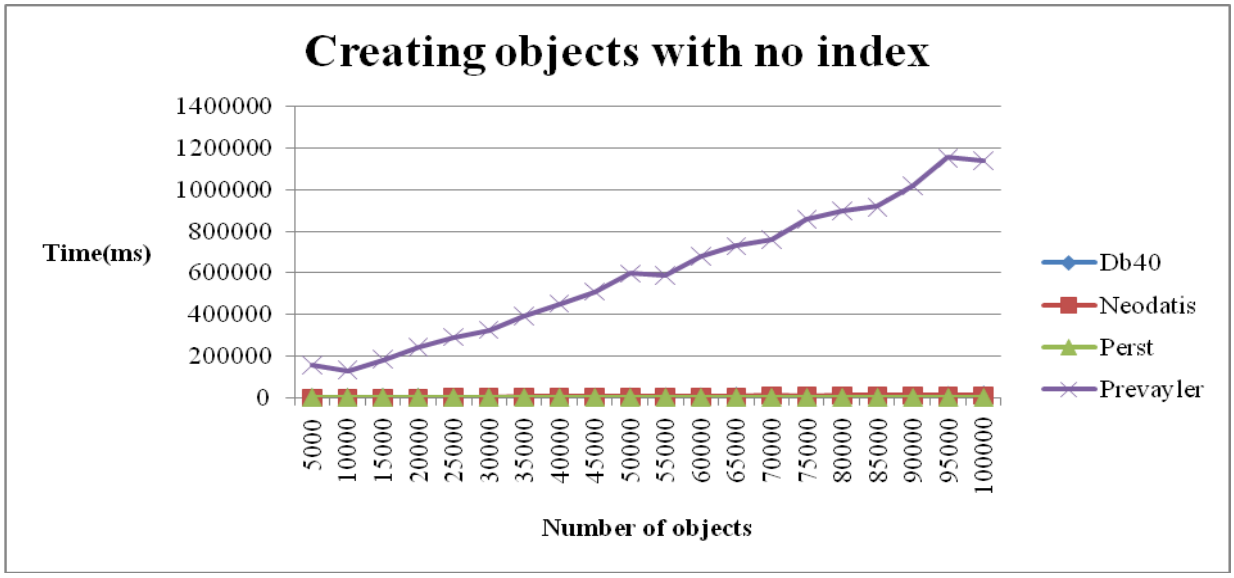

Figure 5.1: Times taken to store objects in the databases

In the graph shown above, the horizontal axis represents the number of objects added in the databases and this ranged from 5000 to 100000. The vertical axis represents the time taken, in milliseconds, to store the objects within the databases. The curves represent the times taken to store objects ranging from 5000 to 100000 within the four databases. The times taken for Prevayler to create objects are higher than for the other databases. Different users of the Prevayler database offered explanations, through the mailing list on their website (Sourceforge 2009), on what the possible causes of this could be. The explanations included:

1. The poor performance of the database may have been caused by other IO operations on the same disc. The remedy suggested for this was to use an entirely separate disc for the database to read and write to.

2. The objects were being inserted into the database one at a time, which was affecting the performance of the database. The suggested remedy was to insert the objects as an array to reduce the time taken to insert each object individually.

Placing the database on a separate disc did not improve its performance to a noticeable degree. The timings were similar, and in some cases, higher than before. Also, the code used to store

the objects was written to enable each object to be added to the database one at a time. The other databases in this study used the same code for storing objects, and therefore it was not desirable to change the way in which objects were inserted.

A possible cause of the high values for Prevayler could be the calling of numerous constructors in its code when creating the database. This could have affected the time that it took to create the different databases.

The curves created by the other databases are barely discernible, as they are closely formed at the bottom of the graph. In order to make an observation of how these curves behaved, another graph is shown below in Figure 5.2:

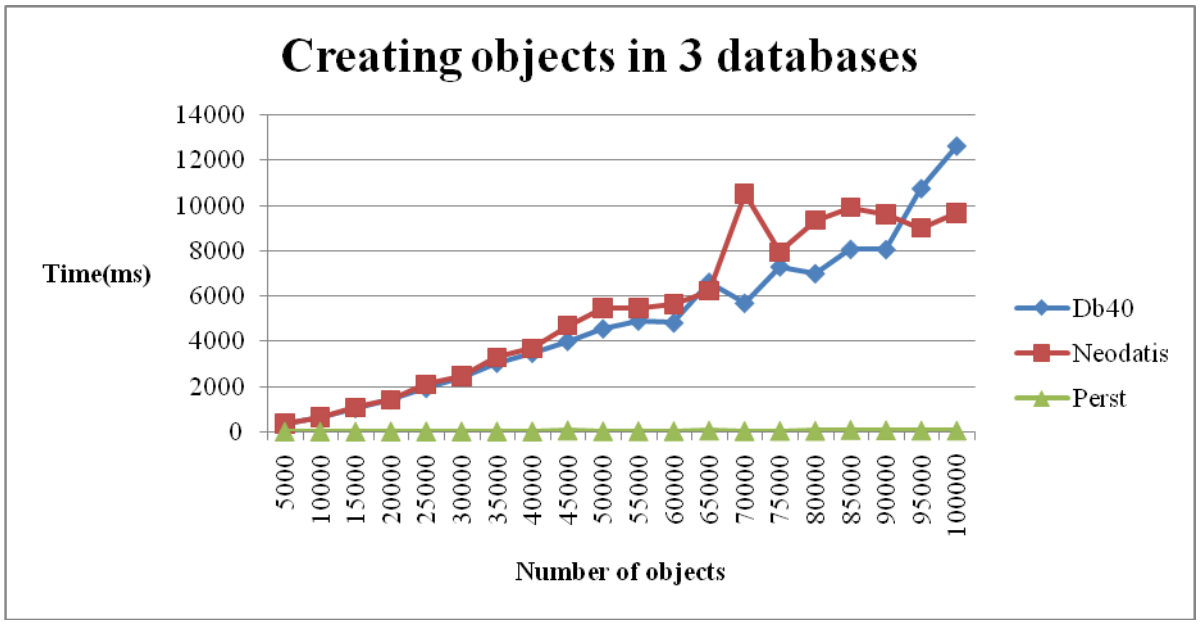

Figure 5.2: Graph showing times taken for storing objects in Db4o, Neodatis and Perst

Db4o and Neodatis exhibit similar performance, though Perst is substantially faster than the other databases.

Neodatis only provides unique indexes and was therefore only able to create indexes on the unique fields of the Item class, namely *unique1*, *unique2*, *stringu1* and *stringu2*. All the other databases that provide indexing were therefore also tested with indexes applied on the four fields. Figure 5.3 below shows the results of this comparison.

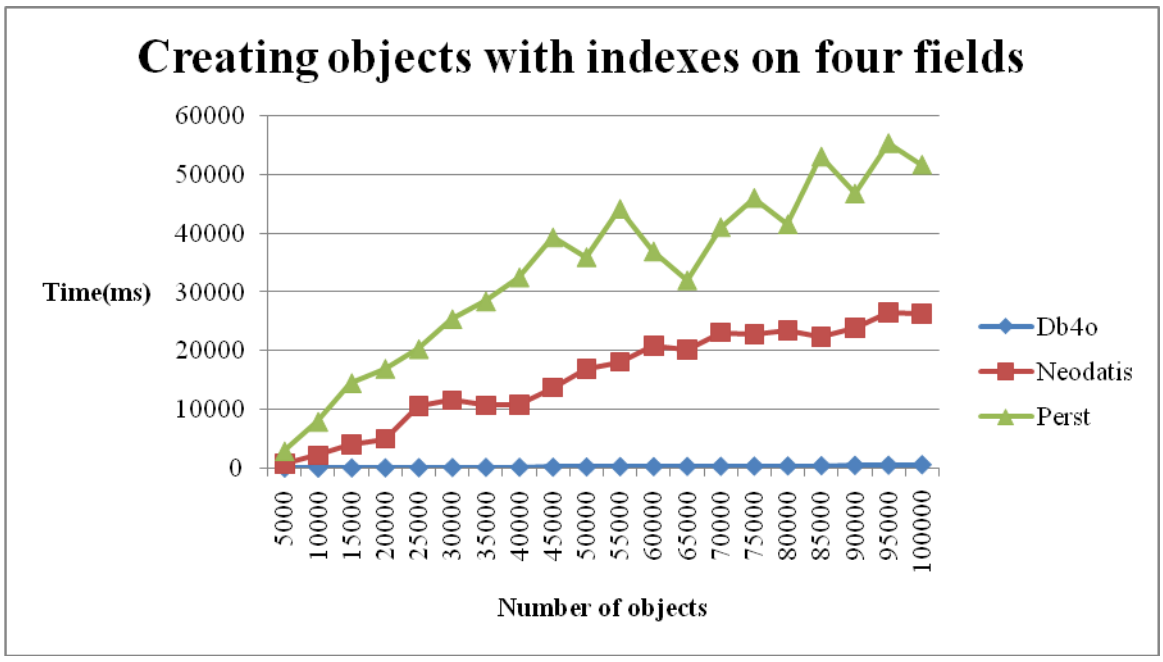

Figure 5.3: Creating objects with indexes applied to four fields

In the figure above, Db4o takes the shortest time and Perst takes the longest time to create objects. It can therefore be concluded that when indexing is applied to the unique fields of the Item class, Db4o takes the shortest time to create objects and Perst takes the longest time, with Neodatis found in between the two databases. Note now that, with indexing, Perst is slowest, whereas it is fastest without indexing. There is a high cost in implementing indexing with Perst. The indexing scheme for Db4o is clearly implemented efficiently.

For all the databases, the amount of time taken to store objects increased as the number of objects increased. In some cases, the amount of time taken by Perst was very small compared to all the databases. It can be concluded that it takes Perst the shortest time with no indexing but the longest time when indexing is enabled.

5.2.2 Querying

5.2.2.1 **Q1. Querying for *unique1***

The values of *unique1* are randomly distributed within the databases. This experiment searched for a single record with *unique1* being equal to 300. Figure 5.4 below illustrates the results obtained.

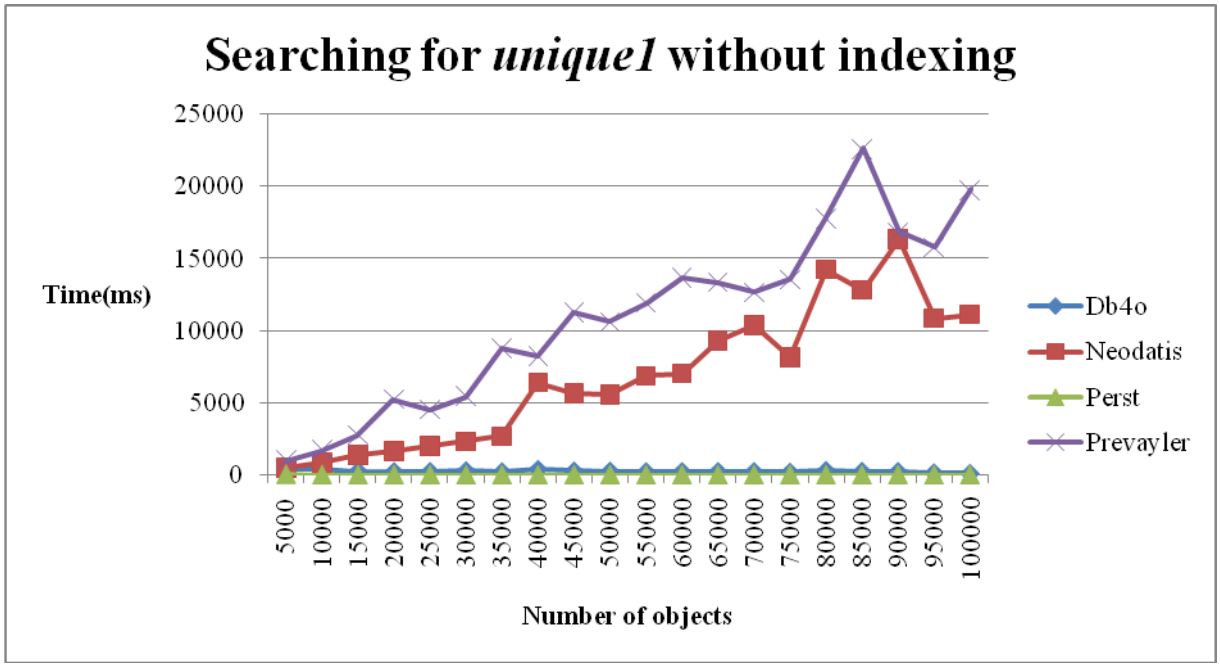

Figure 5.4: Searching for *unique1* without indexing

Prevaylers takes the longest time to search for the values of *unique1*. Perst takes the shortest time, followed by Db4o and Neodatis respectively. It can therefore be concluded from the above figure that Perst performs the search faster.

5.2.2.2 **Q2. Querying for *unique2***

The values of *unique2* are in sequential order from 0, 1, 2, …. (*size* – 1) and are found in the order in which the objects were stored. This experiment searched for the single record with *unique2* equal to 300. The results are shown in Figure 5.5 below.

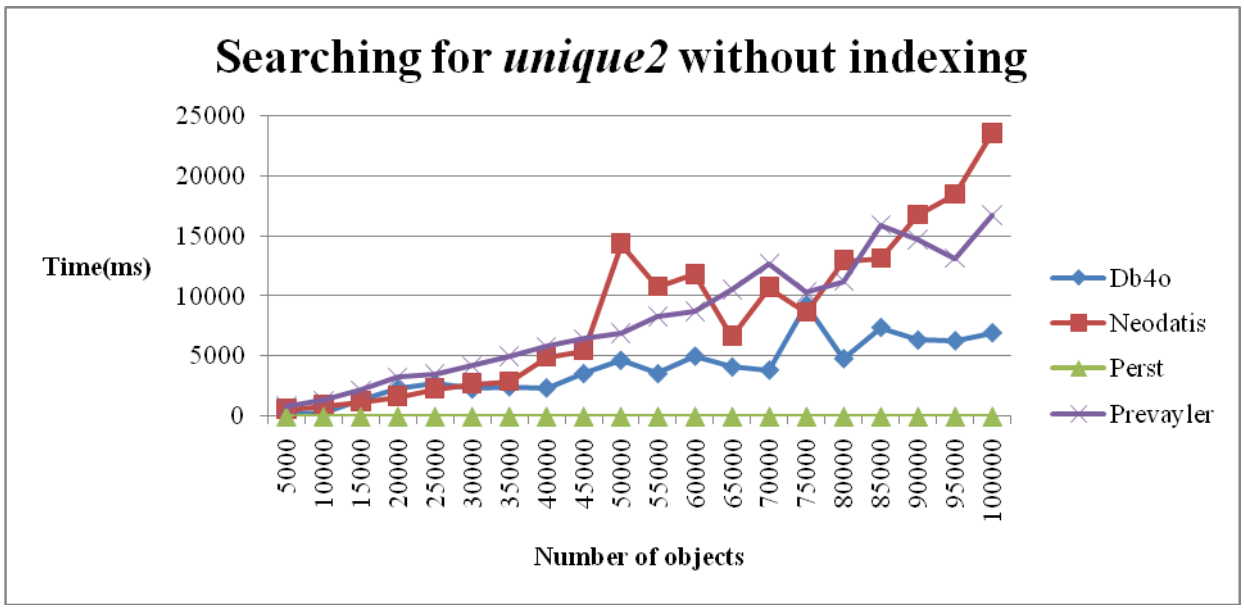

Figure 5.5: Searching for *unique2* without indexing

According to the graph, Neodatis takes the longest time to obtain a result from the query. Neodatis follows with the second highest times, followed by Db4o. Perst takes the shortest time. The standard deviations (SD) obtained when calculating the timings are lowest for Perst and highest for Prevayler. It can therefore be concluded that Neodatis took the longest time to perform the operation and Perst took the shortest time.The times taken by all the databases can be found in table B.2 in Appendix B.

5.2.2.3 **Q3. Querying for one percent of objects, randomly distributed**

The query above, as mentioned in section 4.4.6.2, selects one percent of the objects found in any database. The results of the timings are represented in Figure 5.6 below.

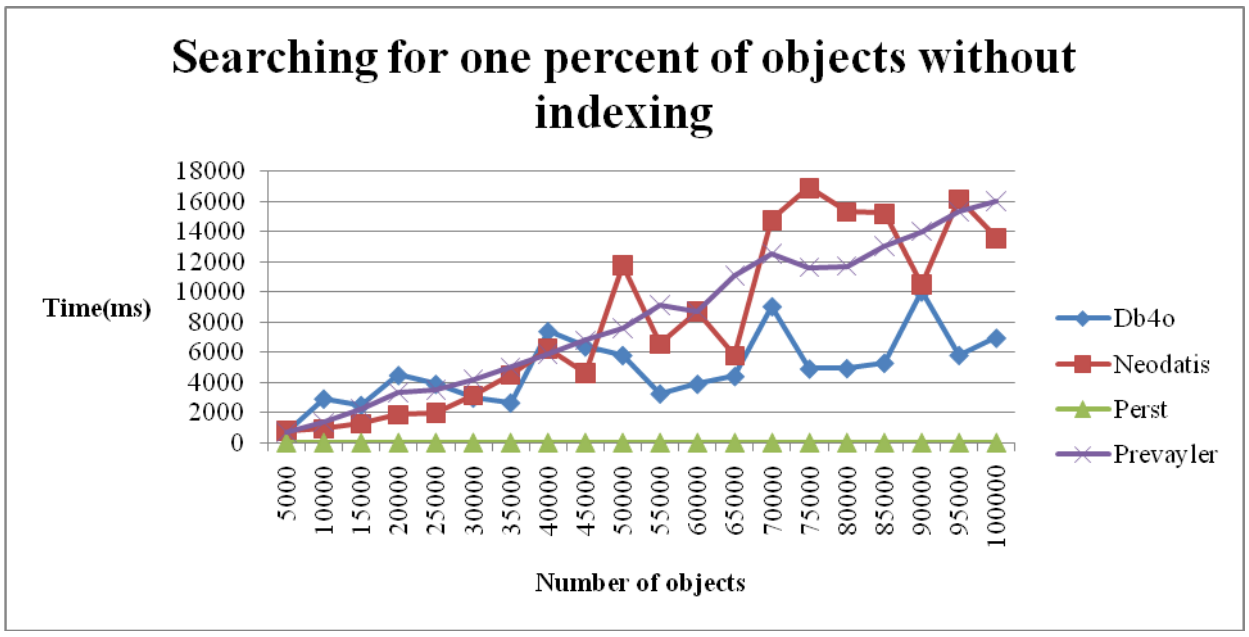

Figure 5.6: Searching for one percent of objects without indexing

Prevayler takes the longest time to perform the search, followed by Neodatis. These are followed by Db4o, with Perst taking the shortest time to perform the query. It can therefore be concluded that Prevayler takes longer in performing the query. The standard deviations (SD) obtained when calculating the timings for Perst was very low, as compared to the other databases, with Db4o's standard deviation being the highest. The times taken by all the databases can be found in table B.3 in Appendix B.

5.2.2.4 **Q4. Querying for ten percent of objects, randomly distributed**

This query searches for ten percent of the objects stored in the database. The results are shown in Figure 5.7 below.

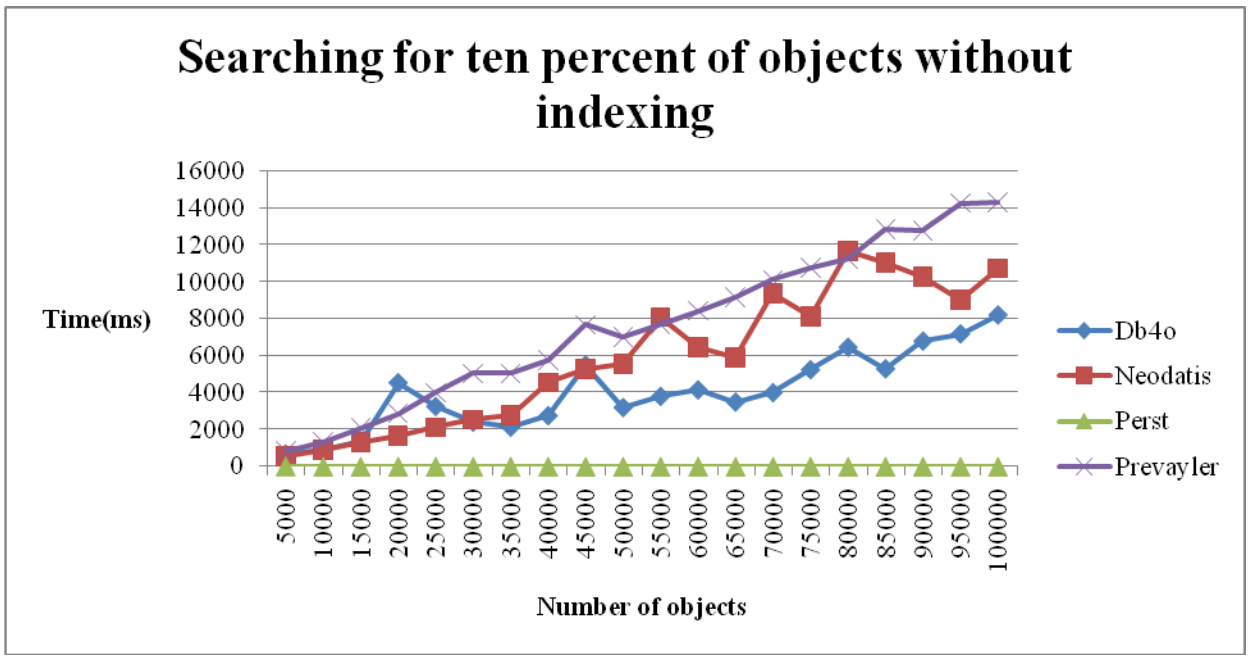

Figure 5.7: Searching for ten percent of objects without indexing

Prevayler takes the longest time, with the other databases performing the query faster than Prevayler. The standard deviation (SD) obtained when doing the timings for Perst is the lowest and that for Neodatis is the highest in all the databases. The times taken by all the databases can be found in table B.4 in Appendix B.

5.2.2.5 **Q5. Querying for twenty percent of objects, randomly distributed**

This query is executed to search for twenty percent of the objects in any particular database. The results are shown in Figure 5.8 below.

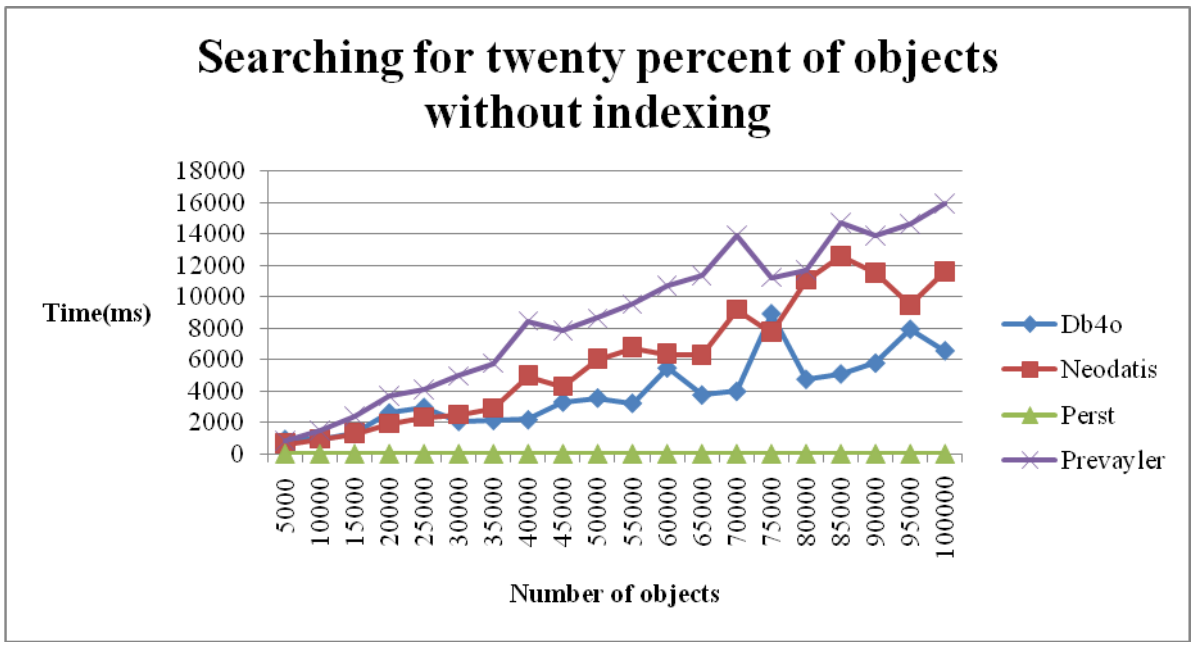

Figure 5.8: Searching for twenty percent of objects without indexing

The results shown above indicate that Prevayler takes the longest time and Perst the shortest in performing the operation. Perst has the lowest standard deviation (SD) and Neodatis has the highest in all the databases. The times taken by all the databases can be found in table B.5 in Appendix B.

5.2.2.6 **Q6. Querying for fifty percent of objects, randomly distributed**

This query selects fifty percent of the objects in any particular size of database. The results are shown in Figure 5.9 below.

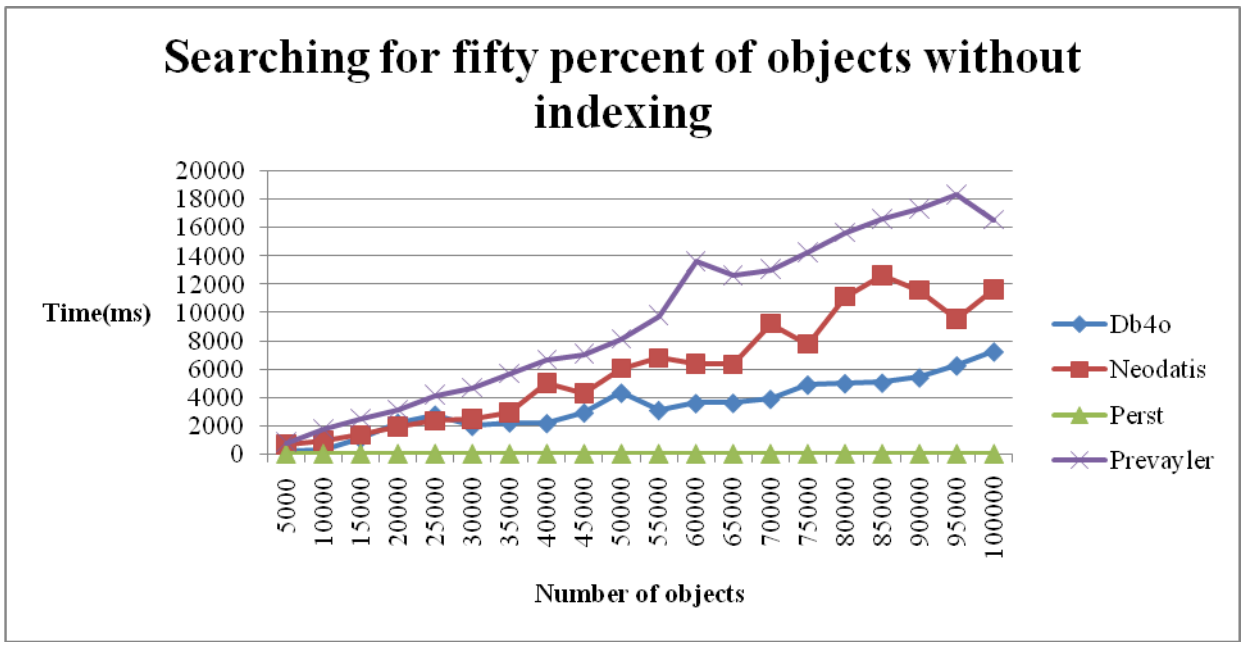

Figure 5.9: Searching for fifty percent of objects without indexing

Prevayler takes the longest time and Perst the shortest time. The standard deviation (SD) for Perst is the lowest calculated and that for Prevayler is the highest across all times recorded. The times taken by all the databases can be found in table B.6 in Appendix B.

From the above experiments, it can be concluded that Prevayler is not particularly desirable for performing searches, whereas Perst performs searches well

5.2.2.7 **Q7. Querying for the first 1000 objects in the database**

This experiment will select all values where *unique2* < 1000. The query will return the first 1000 objects found in the database. The results of the experiment are shown in Figure 5.10 below.

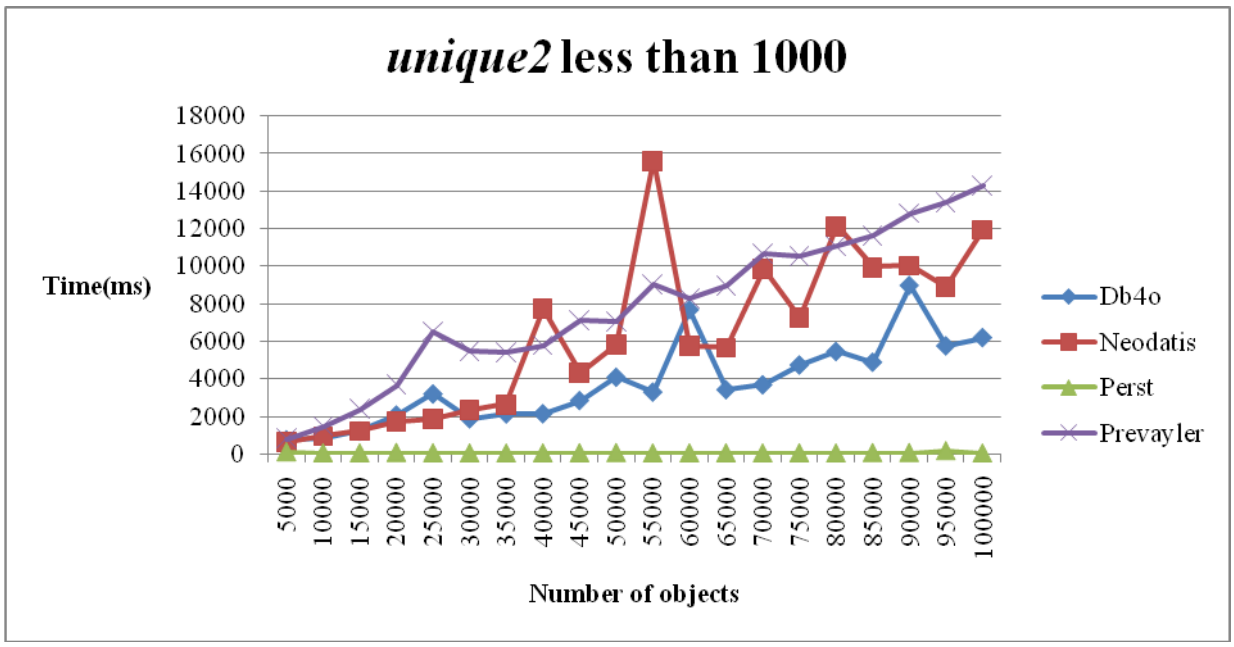

Figure 5.10: Field *unique2* less than 1000

Perst and Db4o take shorter times to perform the experiment respectively as compared to Neodatis and Prevayler. The standard deviation (SD) for Perst is the lowest, followed by Db4o, then Prevayler, and Neodatis comes last with the highest standard deviations calculated. The times taken by all the databases can be found in table B.7 in Appendix B.

5.2.2.8 **Q8. Querying for 1000 randomly dispersed objects**

This query will select all numbers where *unique1* < 1000. This will return 1000 random numbers created in each database. The results are show in Figure 5.11 below.

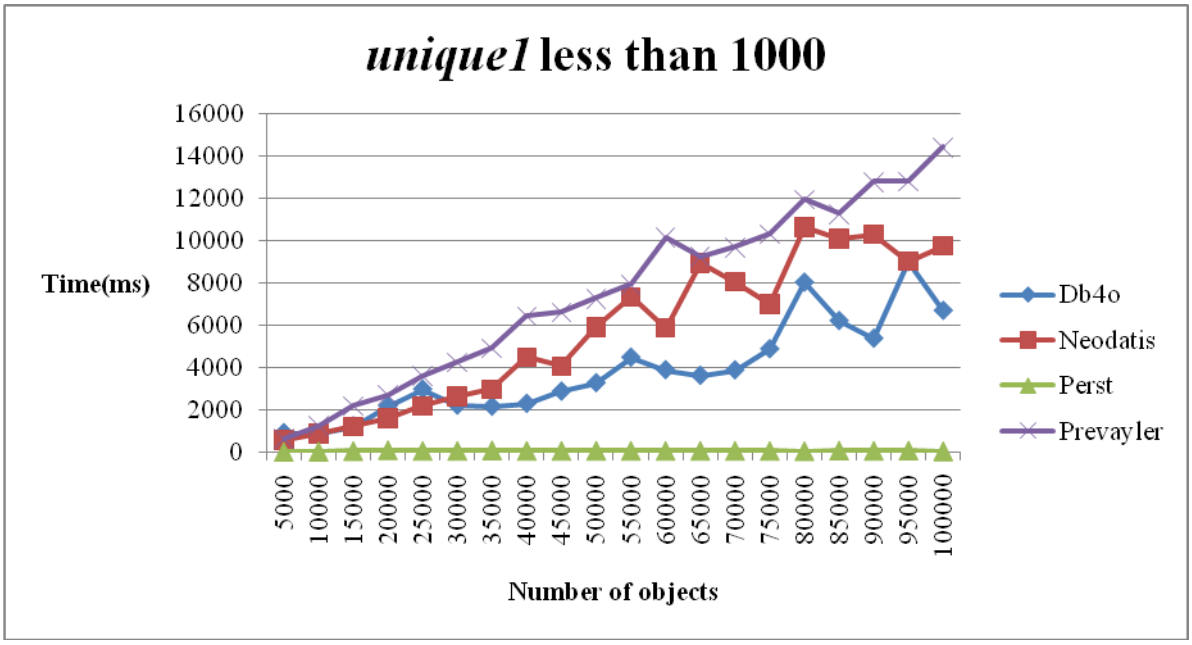

Figure 5.11: Field *unique1* less than 1000

Prevayler takes the longest time to perform the operation, followed by Neodatis. Perst and Db4o take shorter times respectively to carry out the operation. The standard deviation (SD) obtained for the calculations shows that Perst had the lowest standard deviation and Prevayler had the highest in all the databases. The times taken by all the databases can be found in table B.8 in Appendix B.

5.2.2.9 **Q9. Querying strings**

Three experiments were carried out on strings, which involved selecting *stringu1*, *stringu2* and *string4* from the different databases. The results are shown below.

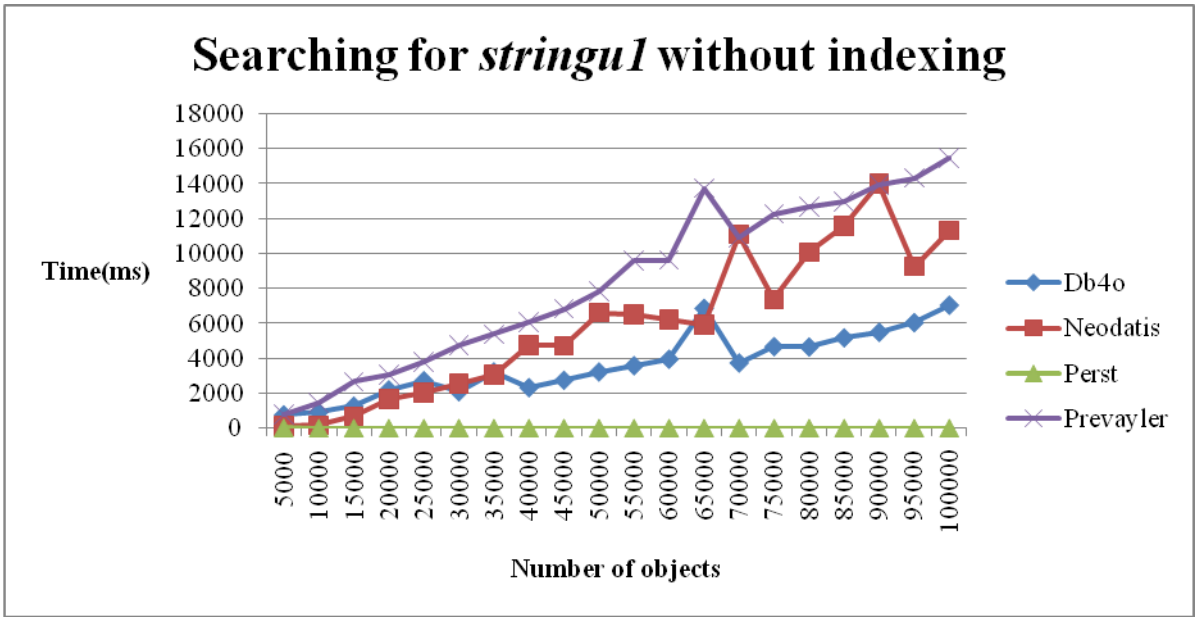

Figure 5.12: Searching for *stringu1* without indexing

The trend that was observed before, of Prevayler taking the longest time to search for values, is repeated in the experiment above. The other databases take shorter times to perform the experiment, with Perst taking the shortest time. Perst has the lowest standard deviation (SD) and Db4o has the highest in all the databases. The times taken by all the databases can be found in table B.9 in Appendix B.

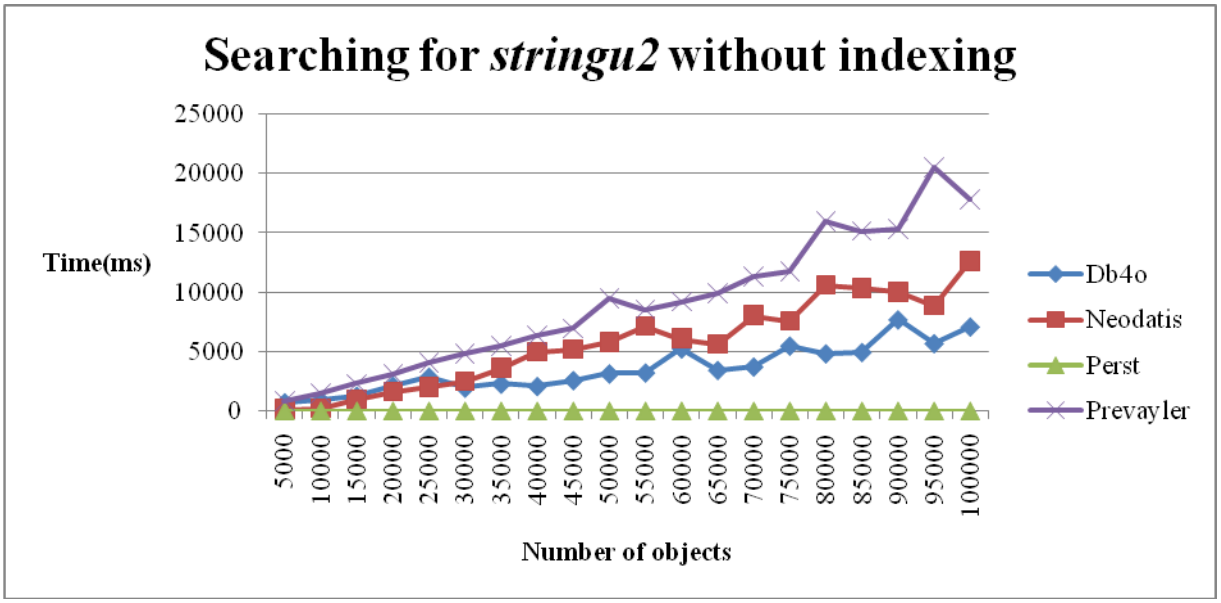

Figure 5.13: Searching for *stringu2* without indexing

Prevayler takes the longest time to perform the above experiment with Perst taking the shortest time. Perst has the lowest standard deviation (SD) and Neodatis has the highest in all the databases. The times taken by all the databases can be found in table B.10 in Appendix B.

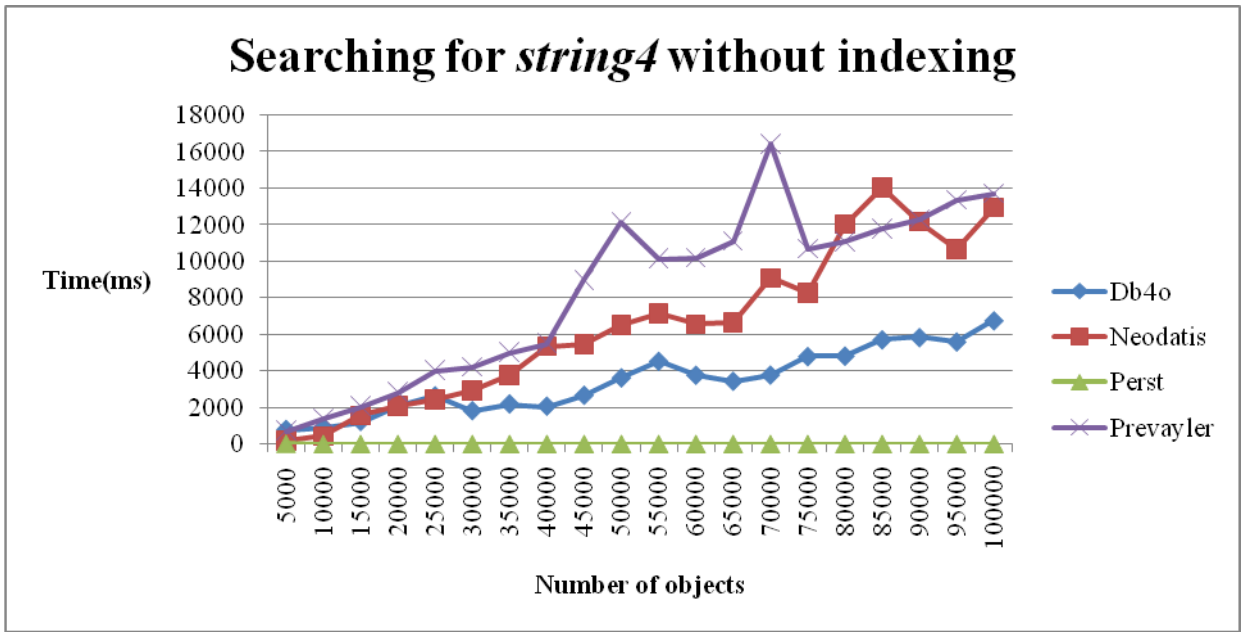

Figure 5.14: Searching for *string4* without indexing

In all the experiments on strings, it can be concluded that Prevayler takes the longest time to perform any particular experiment compared to the other databases. Perst takes the shortest time, with Db4o and Neodatis following respectively in terms of lower timings. The standard deviation (SD) for Perst is the lowet, with that of Neodatis being the highest of all the tested databases. The times taken by all the databases can be found in table B.11 in Appendix B.

5.2.2.10 **Q10. Querying for unique fields with indexing**

As mentioned previously, Neodatis can only apply indexing to unique fields. Prevayler does not provide indexing, and was therefore not considered in the following section. Experiments were carried out on the following fields: *unique1*, *unique2*, *stringu1* and *stringu2*. The graphs shown were created as a result of these experiments.

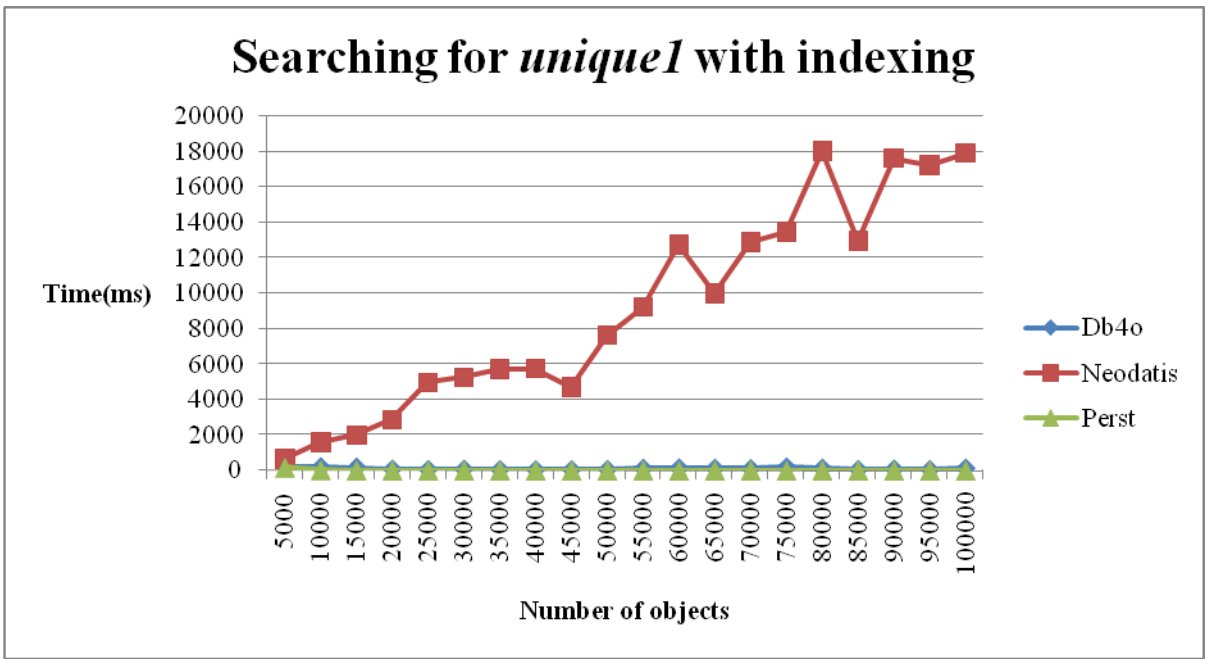

Figure 5.15: Searching for *unique1* with indexing

Db4o and Perst take shorter times to perform the search as compared to Neodatis. The standard deviation (SD) is lowest for Perst and highest for Neodatis. The times taken by all the databases can be found in table B.12 in Appendix B.

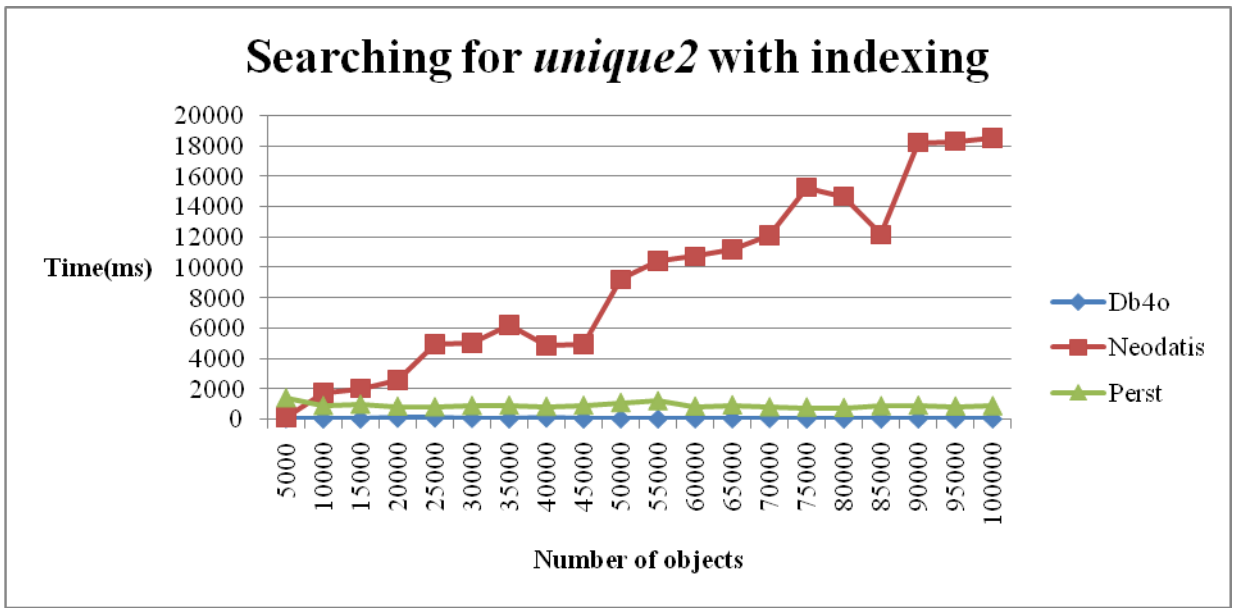

Figure 5.16: Searching for *unique2* with indexing

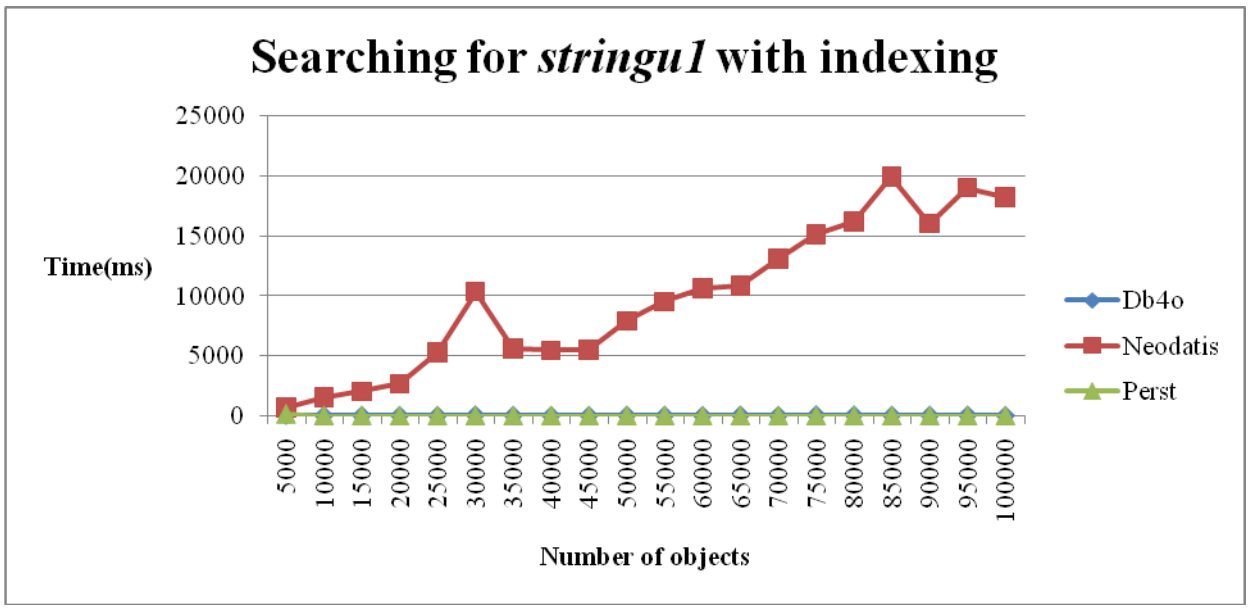

Figure 5.17: Searching for *stringu1* with indexing

The standard deviation (SD) is highest in Neodatis, with Perst and Db4o both having low standard deviations. The times taken by all the databases can be found in table B.13 in Appendix B.

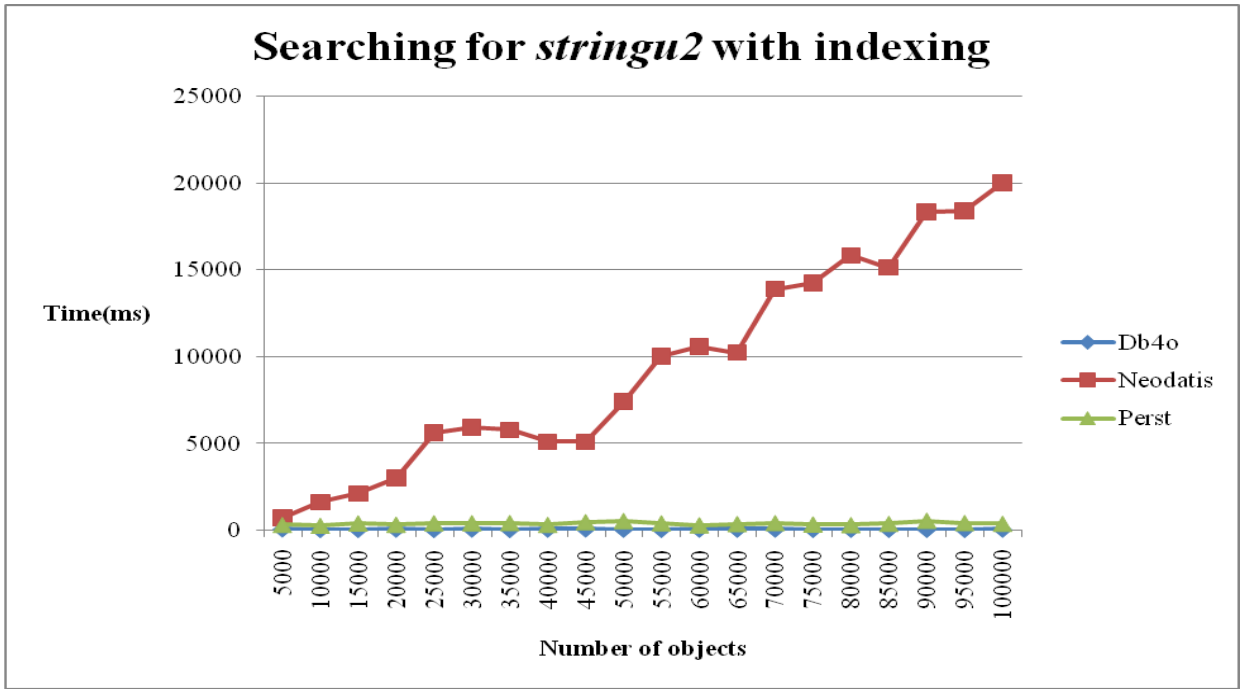

Figure 5.18: Searching for *stringu2* with indexing

Neodatis and Db4o take a similar amount of time to create the index, but the indexing scheme for Db4o is much more efficient when searching. With Perst, the index is created quickly and is also very efficient when searching.

It can be concluded from the above experiments that Neodatis takes the longest time to search for strings as compared to Db4o and Perst.

5.2.3 Updating

The experiments conducted for updating, as mentioned in section 4.4.6.3, involved modifying the Item class by adding a number to the *num* variable. The results of these experiments are shown below.

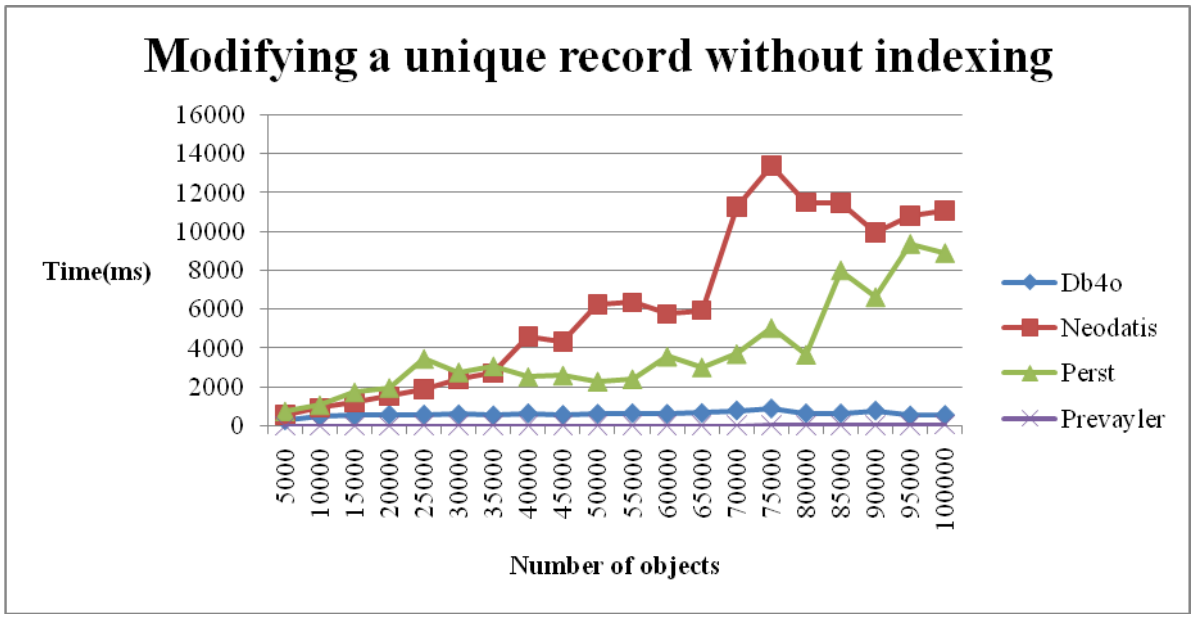

Figure 5.19: Modifying a unique record without indexing

In the figure above, Neodatis takes the longest time to perform the operation, with Perst following it. Prevayler takes the least time and Db4o follows it with the second least time.

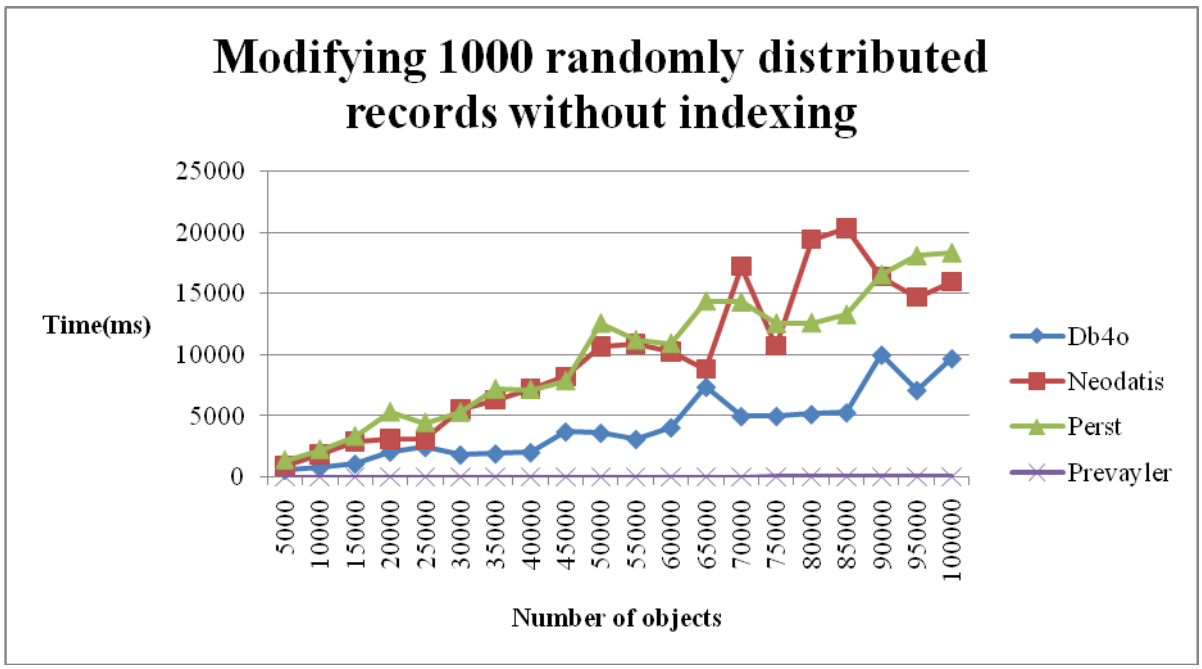

Figure 5.20: Modifying 1000 randomly distributed records without indexing

In Figure 5.20 above, the databases that took the longest times were Perst and Neodatis, which exhibited similar performance. Prevayler performed the updates most efficiently.

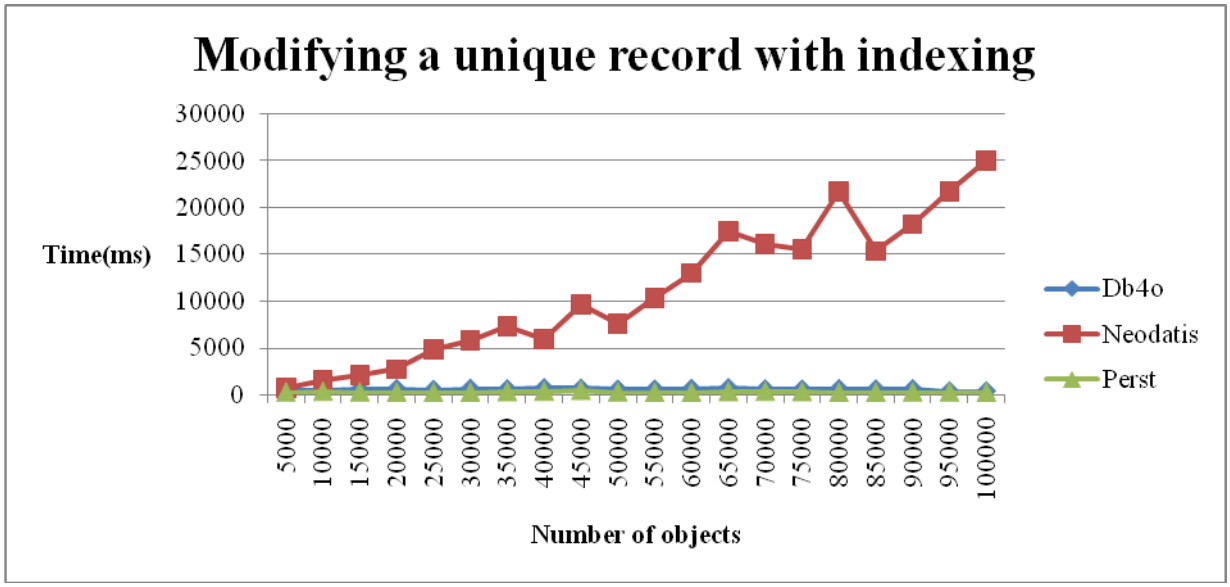

Figure 5.21: Modifying a unique record with indexing

In the figure above, Neodatis again takes the longest time to perform the operation. Perst takes the least time. The standard devition (SD) is very high for all the databases when recording the

timings for this experiement, but Perst has the lowest and Neodatis has the highest. The times taken by all the databases can be found in table B.14 in Appendix B.

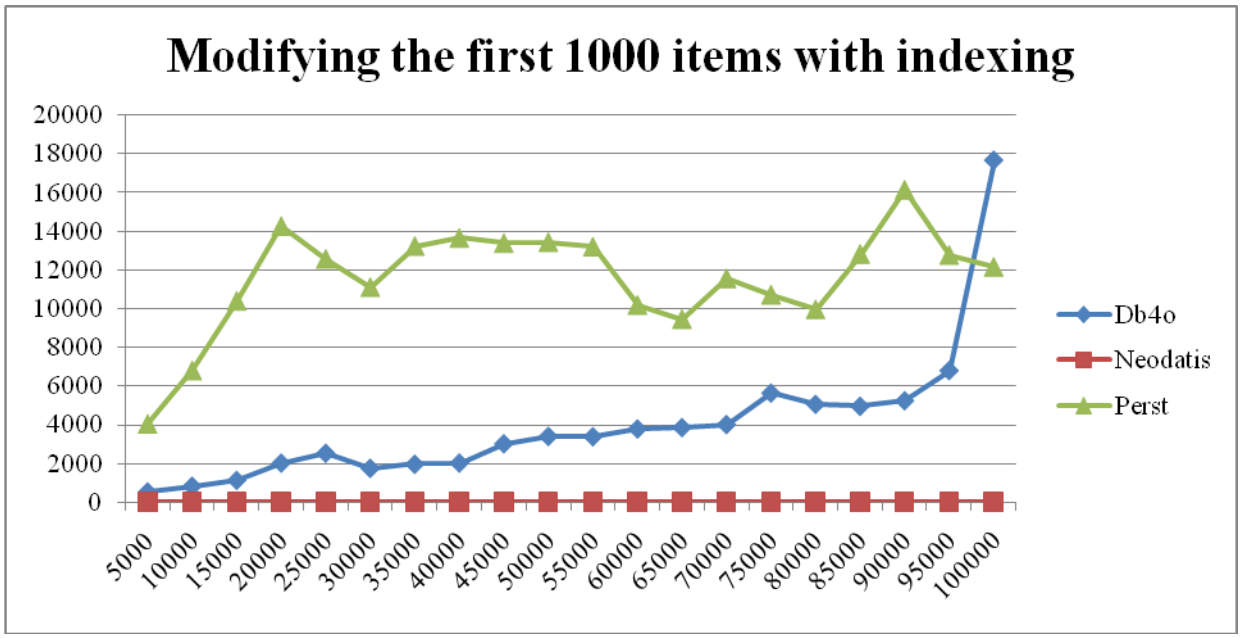

Figure 5.22: Modifying the first 1000 items without indexing

In the figure above, Perst takes the longest times when it is modifying values in the smaller databases. As the databases increase in size, the times taken by Db4o increase. This spike in the graph for Db4o is most likely caused by a timing anomaly. The general trend of the graphs implies that Perst is worse the Db4o. Neodatis takes the least time to perform the experiment.

In most of the experiments performed above, Neodatis takes the most time to perform operations. Perst and Prevayler take the least time, and the timings for Db4o are found in between those of the other databases.

5.2.4 Deleting

The delete operation was performed last. Some experiments were conducted to time this operation for various fields in the database. These results are shown below.

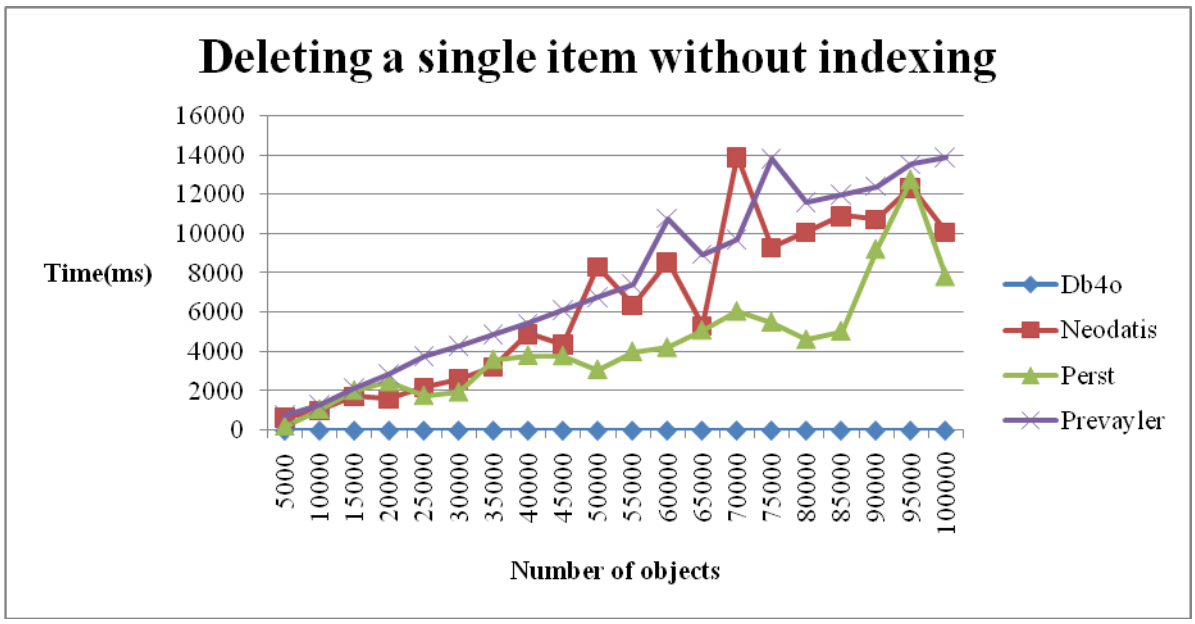

Figure 5.23: Deleting a single item without indexing

In the figure above, Prevayler and Neodatis take longer to perform the operation respectively than Perst and Db4o. The standard deviation (SD) for Db4o is very low, and is 0 in some cases. The SD is highest for Perst. The times taken by all the databases can be found in table B.15 in Appendix B.

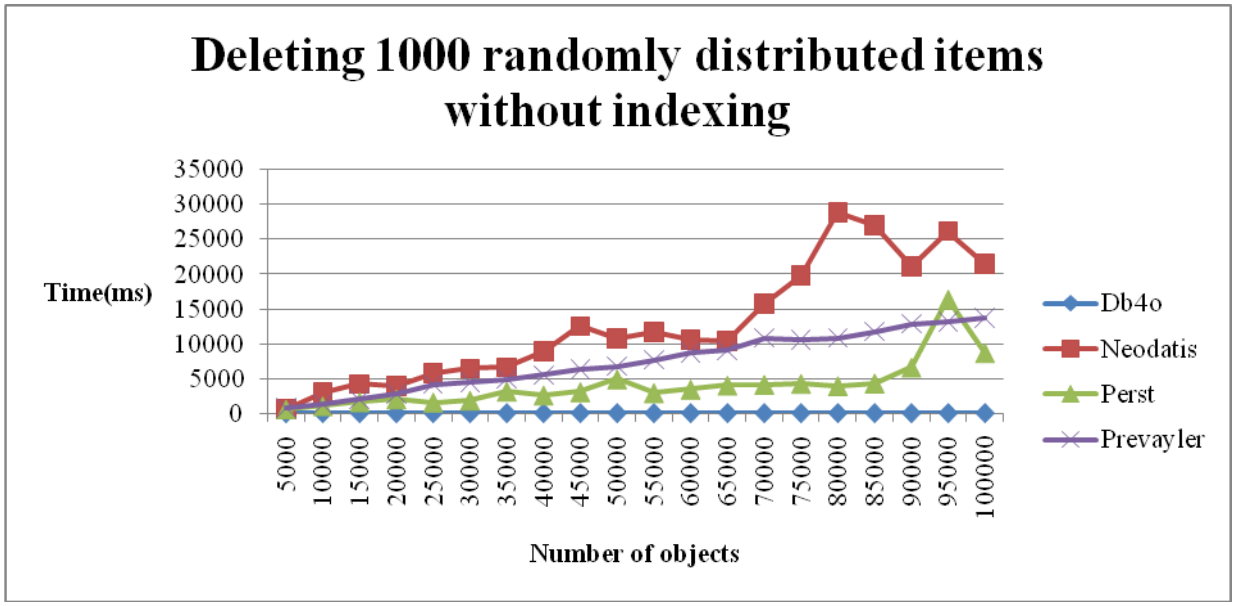

Figure 5.24: Deleting 1000 randomly distributed items without indexing

In the figure above, Neodatis took longer to perform the operation, with Db4o taking the least time. The standard deviation (SD) is also lowest here in Db4o and highest in Neodatis. The times taken by all the databases can be found in table B.16 in Appendix B.

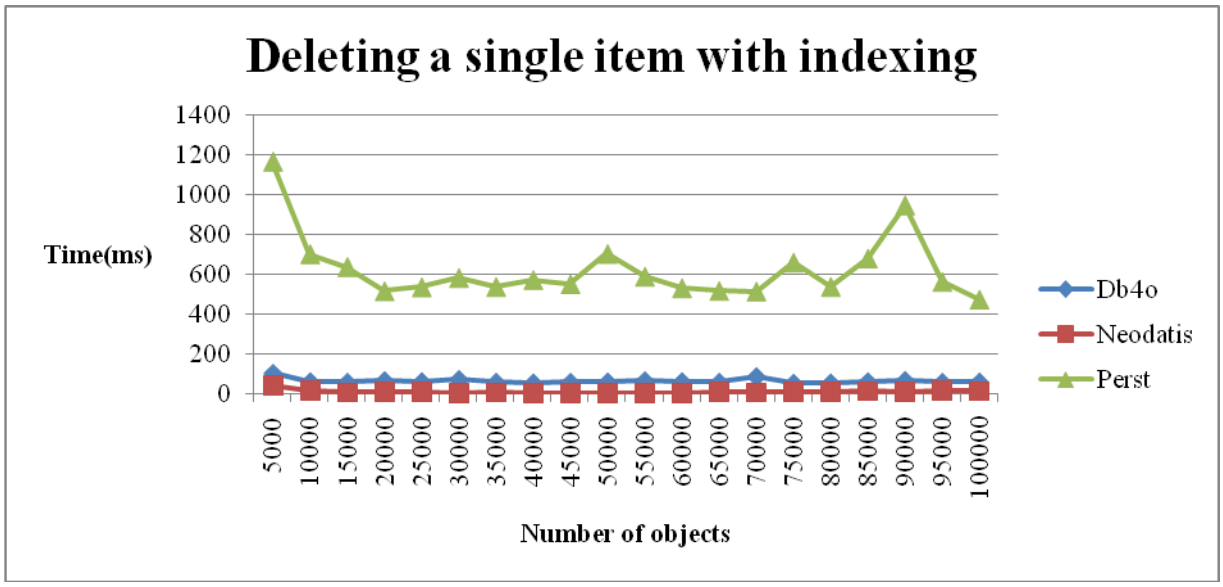

Figure 5.25: Deleting a single item with indexing

In the figure above, Perst takes the longest time to perform the operation. Neodatis takes the least time, followed by Db4o. Comparing figures 5.24 and 5.25, we see that it can take a long time to remove items from the Perst index.

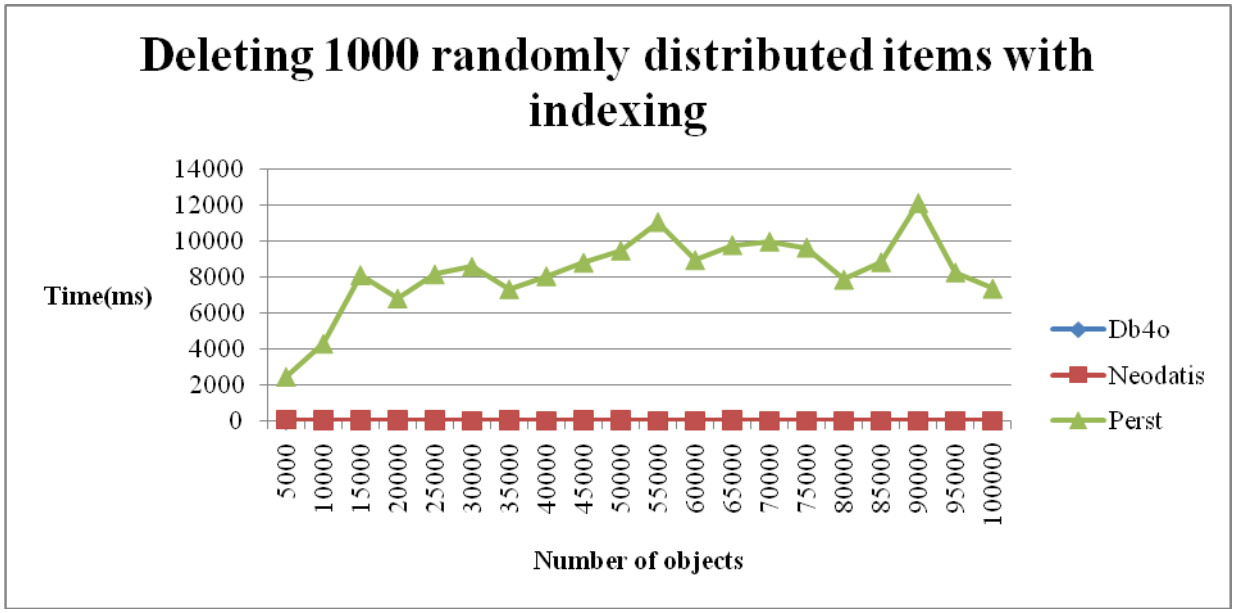

Figure 5.26: Deleting 1000 randomly distributed items with indexing

In the figure above, the graph shows that it took Perst the longest time to perform the operation. Db4o and Neodatis took almost similar times to complete the operation. The curve for Db4o is hardly visible in the figure above and it is therefore not clear that Db4o takes a shorter time than Perst. Looking at all the graphs, it can be concluded that for delete operations, Perst takes the most time and Db4o takes the least. The standard deviation (SD) is lowest for Db4o, followed by Neodatis and Perst comes last with a very high SD. The times taken by all the databases can be found in table B.17 in Appendix B.

The databases above can be ranked in terms of performance, from best to worst, in the table below:

Store no index	**Store with index**	**Query 20% no index**	**Modify no index**	**Modify with index**	**Delete no index**	**Delete with index**
Perst	Db4o	Perst	Prevayler	Perst	Db4o	Neodatis
Db4o	Neodatis	Prevayler	Db4o	Db4o	Prevayler	Db4o
Neodatis	Perst	Db4o	Perst	Neodatis	Perst	Perst
Prevayler	-	Neodatis	Neodatis	-	Neodatis	-

Table 5.1: Databases ranked according to performance

5.3 Discussion of Findings

After all the tests were carried out on the databases, it is possible to make some comparisons on their performance.

Db4o has demonstrated its functionality, as mentioned in section 3.5.2.2. One is able to store, query, update and delete database objects. Db4o also provides indexing and is scalable, with as many as 100000 objects being added to the database. Db4o was able to excel in tasks involving deleting of objects from the databases.

Neodatis performed many of the operations the slowest. Neodatis has proven its functionality, as mentioned in section 3.5.3.2. Objects were added, queried, updated and deleted. Neodatis also provided indexing. Neodatis took a short time to create objects, but was the database that lagged behind when it came to many of the other database operations. Db4o and Perst fell in between these two extremes of performance.

Perst performs consistently, with its times for creating, searching and updating being fast. The only time that Perst seems to suffer in its performance is when it has to carry out the delete operation. Perst has good functionality, as mentioned in section 3.5.4.2. It was able to store large volumes of data very fast. This also serves to confirm its scalability, as mentioned in section 3.5.4.8.

Prevayler took a long time to create objects. It also took longer times when it came to performing searches. This is in contrast to the claim by Prevayler, as shown in section 3.5.5.2, that their database is simple and fast. The code segments and documentation provided were inadequate. On the other hand, Prevayler performs modification of objects very fast.

A possible reason for Prevayler behaving the way is does is that the developers may have invested a lot of time in ensuring that the creation of the database will speed up all the other

functions. Usually, one part of a product's performance may suffer in order to accommodate it being superior in another part.

By referring to the results above, a user may be able to decide which of the databases to use for their requirements.

5.4 Conclusion

This chapter covered the experiments carried out on all the databases. Various experiments were conducted in order to test on the speed of these databases in producing results. The graphs created from the experimental results can be used to make an informed decision on what database to use depending on the user's requirements. A user that would want a fast database for creating or storing objects should use Perst or Db4o, rather than Prevayler. On the other hand, if a database is needed that will perform searches, updates and deletes, Prevayler may well be a user's best option.

CHAPTER 6

CONCLUSION AND FUTURE WORK

6.1 Introduction

This chapter will conclude the project on the comparison of Open source object-oriented database products. It will provide a summary of the project, a summary of the findings and discuss the conclusions reached from these findings. A summary of the conclusions and suggestions for future research will also be covered.

6.2 Project Summary

The project involved comparing the performance of four Open Source object-oriented database products on their ability to perform the basic database operations. These operations are creating, querying, updating and deleting database objects.

The selected databases were Db4o, Neodatis, Perst and Prevayler. Java code was written in order to test these databases. The tests were carried out and timings were taken that would show which of the databases performed a particular function the fastest.

These timings were recorded and used to create graphs that showed the performance of these databases.

6.3 Work Covered

The experiments that were carried out were structured in such a way as to enable the author to confirm whether the object databases were able to perform the standard database operations. Graphs were used to show how these different databases performed the experiments.

The thesis statement in chapter one was as follows:

- Some Open Source object-oriented databases are better than others at performing the standard database operations of storing, querying, updating and deleting database objects.

The results obtained were able to confirm that these Open Source object-oriented databases are able to perform the standard database operations of storing, querying, updating and deleting database objects.

6.4 Conclusions

The following conclusions were made at the end of the project:

- All the databases used in this study are able to carry out the basic standard operations of creating, querying, updating and deleting database objects.

- Prevayler was the slowest database and Perst was the fastest when it came to creating objects.

- Perst was the fastest in searching and updating, but was slowest when it came to deleting objects from the databases.

- Neodatis performed many of the operations the slowest, apart from the operation for creating objects.

- Db4o performed operations at different speeds compared to the other databases, but it was neither the slowest nor fastest at completing any operation.

- Not all the databases offered indexing.

- Object databases can prove difficult to use if one does not have the necessary programming skills.

- The performance of the different databases made them suitable for different operations. For example, in a case where searching is more important, like in a library, Perst would be more suitable to be used as it performs searches faster than any of the other

databases. Neodatis would be suitable for a case where a user would want to enter objects into the database quickly and not need to search for them very often, such as a payroll system.

6.5 Summary of Contributions

The conclusions reached in section 6.4 above may serve as a guide to anyone who may want to use the chosen object-oriented databases. The experiments carried out on these databases showed which was suited to perform a particular function.

6.6 Suggestions for Further Research

Some suggestions for future work may include, but are not limited to, the following:

- The number of databases can be increased in order to broaden the scope of products being tested.

- The number objects being added to the databases may be increased to more than 100000 so that the behaviour of these databases in terms of scalability and performance can be tested.

- Newer versions of the object databases can also be used to run similar experiments.

- The number of experiments may be increased in order to investigate different scenarios when dealing with object databases.

6.7 Conclusion

Open Source object-oriented databases are found in our everyday lives. Their use is gaining popularity and though it may not surpass that of the commonly used relational databases, an understanding of their functioning and mechanics is needed in order to appreciate them.

REFERENCES

Ambler, S.W. 2008. **Introduction to Concurrency Control**. Available at: http://www.agiledata.org/essays/concurrencyControl.html [Accessed: October 23 2008].

Apple.com. 2008. **Introduction to Performance Overview**. Available at: http://developer.apple.com/documentation/Performance/Conceptual/PerformanceOverview/DevelopingForPerf/chapter_2_section_2.html [Accessed: November 12 2008].

Atkinson, M., Bancilhon, F., DeWitt, D., Dittrich, K., Maier, D., Zdonik S. 2003. **The Object-Oriented Database System Manifesto**. Available at: http://www.cl.cam.ac.uk/teaching/2003/Databases/oo-manifesto.pdf [Accessed: 12 September 2008].

Berre, A.J., Anderson, T.L., Mallison, M. **The HyperModel Benchmark. Technical Report No. CSE 88-031**. Oregon Graduate Institute, Department of Computer Science. USA.

Bitton, D., DeWitt, D.J., Turbyfill, C. **Benchmarking Database Systems: A Systematic Approach**. Proceedings of the 1983 VLDB (Very Large Data Bases) Conference.

Carey, M.J., DeWitt, D.J., Naughton, J.F. **The 007 Benchmark**. *ACM SIGMOD Record*, Volume 22, (2) June 1993. P 12-21.

Casson, A. 1994. **What are Object Oriented Databases?** Available at: http://www.aiai.ed.ac.uk/project/plinth/oodb/what.html [Accessed: October 29 2008].

Chaudhri, A.K., Zicari, R. 2001. **Succeeding with Object Databases – A Practical Look at Today's Implementations with Java ™ and XML**. John Wiley & Sons, Inc. Canada.

Codehaus.org. 2004. **Object Query Languages**. Available at: http://docs.codehaus.org/display/PREVAYLER/Object+Query+Languages [Accessed: November 17 2008].

Codehaus.org. **Prevayler Development Team**. Available at: http://prevayler.codehaus.org/Team [Accessed: October 22 2008].

Colorado State University. 2009. **Advantages and Disadvantages of Experimental Research**. Available at: http://writing.colostate.edu/guides/research/experiment/pop5c.cfm [Accessed: January 16 2009].

Db4o user guide. Available with product download.

DeWitt, D. 1993. **The Wisconsin Benchmark**. Available at: http://firebird.sourceforge.net/download/test/wisconsin_benchmark_chapter4.pdf [Accessed: January 27 2009].

DeWitt, D.J. 1993. ***The Wisconsin Benchmark: Past, Present, and Future*. The Benchmark Handbook for Database and Transaction Systems (2nd Edition)**. Morgan Kaufmann Publishers, USA.

Elmasri, R., Navethe, S.B. 1994. **Fundamentals of Database Systems Second Edition**. The Benjamin/Cummings Publishing Company Inc. Redwood City, CA. 94065.

Freshmeat.net. 2009. Available at: http://freshmeat.net [Accessed: 30 September 2009].

Freshmeat.net. 2009. **Freshmeat measure of popularity**. Available at: http://help.freshmeat.net/faqs/statistics/how-do-you-measure-a-projects-popularity [Accessed: 30 September 2009].

Freshmeat.net. 2009. **Freshmeat measure of vitality**. Available at: http://help.freshmeat.net/faqs/statistics/how-do-you-measure-a-projects-vitality [Accessed: 30 September 2009].

Freshmeat.net. 2009. **Prevayler review on Freshmeat**. Available at: http://freshmeat.net/projects/Prevayler [Accessed: 30 September 2009].

Geeknet Inc. 2009. **Neodatis ODB: Topics for Help**. Available at: http://sourceforge.net/forum/forum.php?forum_id=619814 [Accessed: November 17 2008].

Geeknet, Inc. 2009. **Mailing Lists for Prevayler**. Available at: http://sourceforge.net/mail/?group_id=36113 [Accessed: November 24 2008].

Gray, J.N. 1993. **The Benchmark Handbook for Database and Transaction Processing Systems**. Morgan Kaufmann Publishers, USA.

Hughes, J.G. 1991. **Object-Oriented Databases**. Prentice Hall International (UK) Ltd. United States of America.

InterSystems Corp. 2008. Available at: http://vista.intersystems.com/csp/docbook/DocBook.UI.Page.cls?KEY=GOBJ_oo [Accessed: October 24 2008].

Jenkov, J. **Java Concurrency: Deadlock Prevention**. Available at: http://tutorials.jenkov.com/java-concurrency/deadlock-prevention.html [Accessed: 18 September 2009].

Jenkov, J. **Java Concurrency: Deadlock**. Available at: http://tutorials.jenkov.com/java-concurrency/deadlock.html [Accessed: 18 September 2009].

Khayundi, P., Chadwick, J. **A Comparison of Open Source Object-Oriented Database Products**. In Proceedings of SATNAC (Southern African Telecommunication Networks & Appliances Conference) 2008, Wild Coast, South Africa. September 2008.

Korth, H.F., Silberschatz, A. 1986. **Database System Concepts**. McGraw-Hill, Inc. USA.

McObject.com. 2009. **McObject Management**. Available at: http://www.mcobject.com/management [Accessed: November 17 2008].

McObject.com. 2009. **Perst Features and Benefits**. Available at: http://www.mcobject.com/perst_features_benefits [Accessed: November 30 2008].

McObject.com. 2009. **Perst Forum**. Available at: http://forums.mcobject.com/index.php?showforum=4 [Accessed: November 17 2008].

McObject.com. 2009. **Perst Introduction and Tutorial**. Available at: http://www.mcobject.com/index.cfm?fuseaction=download&pageid=457&sectionid=114 [Accessed: November 17 2008].

McObject.com. 2009. **Perst Product Website**. Available at: http://www.mcobject.com/perst [Accessed: November 17 2008].

McObject.com. 2009. **Perst Target Markets**. Available at: http://www.mcobject.com/markets [Accessed: November 18 2008].

McObject.com. 2009. **Users of McObject Embedded Databases**. Available at: http://www.mcobject.com/who_uses_mcobject1 [Accessed: November 17 2008].

Mobixess Inc. 2009. **JODB-The Free Object Database for Java**. Available at: http://www.java-objects-database.com/ **[**Accessed: July 15 2009].

MyOoDB. **Myoodb Developer Information**. Available at: http://www.myoodb.org/aboutauthor.html [Accessed: July 15 2009].

N. Mabanza, J. Chadwick. **A comparison of Open Source XML Database Products**, Proceedings SATNAC conference, Stellenbosch, South Africa, 2006. ISBN 0-620-37043-2.

N. Mabanza, J. Chadwick. **Performance evaluation of Open Source Native XML databases – A Case Study**, Proceedings of IEEE - the 8th International Conference on Advanced Communication Technology (ICACT), Phoenix Park, Korea. ISBN 89-5519-130-8**.**

Neodatis user guide. 2008. Available with product download.

Neodatis.org, 2009. **Neodatis Team**. Available at: http://www.neodatis.org/team [Accessed: October 20 2008].

Neodatis.org. 2009. **Neodatis Overview**. Available at: http://www.neodatis.org/overview [Accessed: October 20 2008].

Neodatis.org. 2009. **Neodatis Product Website**. Available at: http://www.neodatis.org [Accessed: November 17 2008].

Neodatis.org. 2009. **Users of Neodatis**. Available at: http://www.neodatis.org/whos-is-using [Accessed: October 20 2008].

Obasanjo, D. 2001. **An Exploration of Object Oriented Database Management Systems**. Available at: http://www.25hoursaday.com/WhyArentYouUsingAnOODBMS.html [Accessed: September 1 2008].

Ozone-db.org. 2009. **Ozone Open Source Java Object Database Management System**. Available at: http://www.ozone-db.org/frames/about/about.html **[**Accessed: July 14 2009].

Panel Discussion: Database system performance management. **Proceedings of the 1986 ACM SIGMOD International Conference on Management of Data.** Washington, D.C., United States. p 153 – 154.

PC Mag.com. 2009. **JSQL Definition**. Available at: http://www.pcmag.com/encyclopedia_term/0.2542,t=JSQL&i=45688,00.asp [Accessed: November 17 2008].

Pratt, J.P., Adamski, J.J. 2005. **Concepts of Database Management, Fifth Edition.** Thomson Learning Inc. United States of America.

Prevayler.org. 2006. **Prevayler Pioneers and Users**. Available at: http://www.prevayler.org/old_wiki/PrevaylerPioneers.html [Accessed: November 17 2008].

Prevayler.org. 2007. **Prevayler Features**. Available at: http://www.prevayler.org/wiki [Accessed: November 17 2008].

Prevayler.org. 2007. **Prevayler Product Website**. Available at: http://www.prevayler.org/wiki/;jsessionid=95305B805BA647C47B2F0C019D8FE63F [Accessed: November 30 2008].

Rolland, F.D. 1998. **The Essence of Databases**. Prentice Hall, United Kingdom.

Seng, J., Yao, S.B., Hevner, A.R. 2003. **Requirements-driven database systems benchmark method**. Available at: www.sciencedirect.com. [Accessed: 4 November 2009].

Seng, Jia-Lang. 1998. **Comparing Object-Oriented Database Systems Benchmark Methods**. In Proceedings of the Thirty-First Annual Hawaii International Conference on System Sciences, Volume 6, Page 455.

Sourceforge.net 2009. PolePosition benchmark results. Available at: http://polepos.sourceforge.net/results/PolePosition.pdf [Accessed 4 November 2009].

Sourceforge.net 2009. **PolePosition Open Source Database Benchmark**. Available at: http://polepos.sourceforge.net/ [Accessed: 22 October 2009].

Spencer, D. 2004. **What is usability?** Available at: http://www.steptwo.com.au/papers/kmc_whatisusability [Accessed: September 24 2008].

Techtarget.com. 2009. **Definition of Functionality**. Available at: http://searchsoa.techtarget.com/sDefinition/0,,sid26_gci335477,00.html [Accessed: October 20 2008].

Techtarget.com. 2009. **Definition of Support**. Available at: http://whatis.techtarget.com/definition/0,,sid9_gci774434,00.html [Accessed: October 20 2008].

Transaction Processing Performance Council. 1992. **TPC Benchmark A**, Standard Specification. Available at: http://www.tpc.org/tpca/default.asp [Accessed: 22 October 2009].

Transaction Processing Performance Council. 1992. **TPC Benchmark B**, Standard Specification. Available at: http://www.tpc.org/tpcb/default.asp [Accessed: 22 October 2009].

Transaction Processing Performance Council. 1995. **TPC Benchmark C**, Standard Specification. Available at: http://www.tpc.org/tpcc/default.asp [Accessed: 22 October 2009].

Transaction Processing Performance Council. 1998. **TPC Benchmark D**, Standard Specification. Available at: http://www.tpc.org/tpcd/default.asp [Accessed: 22 October 2009].

Transaction Processing Performance Council. 2001. **TPC Benchmark W**, Standard Specification. Available at: http://www.tpc.org/tpcw/default.asp [Accessed: 22 October 2009].

Van Zyl, P., Kourie, D. G., Boake, A. 2006. **Comparing the Performance of Object Databases and ORM Tools.** Proceedings of SAICSIT 2006 Annual Conference of the South African Institute of Computer Scientists and Information Technologists, 9–11 October 2006, Pretoria, South Africa.

Versant Corp. 2008. **Db4o discussion forums**. Available at: http://developer.db4o.com/forums/ [Accessed: October 20 2008].

Versant Corp. 2009. Available at: http://www.db4o.com/s/benchmarkdb.aspx [Accessed: 22 October 2009].

Versant Corp. 2009. Available at: http://www.db4o.com/about/productinformation/db4o/ [Accessed: November 26 2008].

Versant Corp. 2009. **Db4o Product Website**. Available at: http://www.db4o.com [Accessed: October 22 2008].

Wheeler, D.A. 2009. **How to Evaluate Open Source Software/Free Software (OSS/FS) Programs**. Available at: http://www.dwheeler.com/oss_fs_eval.html **[**Accessed: September 22 2009].

Wulf, W.A. 2001. **What is performance?** Available at: http://java.sun.com/docs/books/performance/1st_edition/html/JPPerformance.fm.html [Accessed: November 12 2008].

APPENDICES

Appendix A

A.1. Java Code for Item Class

A.1.1. Item.java

```
// This code creates an item to be used to perform the experiments

public class Item implements java.io.Serializable{
	int num;
	int unique1, unique2;
	int onePercent, tenPercent;
	int twentyPercent, fiftyPercent;
	String stringu1, stringu2, string4;

	public Item(int unique1, int unique2) {
	/*************************************/
		String[] list4 = {"AAAA", "HHHH", "OOOO", "VVVV"};
		String x45 = "xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx";
		this.unique1 = unique1;
		this.unique2 = unique2;
		this.onePercent = unique1 % 100;
		this.tenPercent = unique1 % 10;
		this.twentyPercent = unique1 % 5;
		this.fiftyPercent = unique1 % 2;
		this.stringu1 = convert(unique1) + x45;
		this.stringu2 = convert(unique2) + x45;
		this.string4 = list4[unique2%4] + x45 + "xxx";
		}

	public Item (int num) {
		this.num = num;
	}

	public Item(String str) {
		String x45 = "xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx";
		this.unique1 = 0;
		this.unique2 = 0;
		this.onePercent = unique1 % 100;
		this.tenPercent = unique1 % 10;
		this.twentyPercent = unique1 % 5;
		this.fiftyPercent = unique1 % 2;
		this.stringu1 = str+x45;
		this.stringu2 = null;
```

```
            this.string4 = null;
        }
         private String convert(int n) {
        /*****************************/
            char[] result = new char[7];
            char[] temp = new char[7];
            for(int k = 0; k < 7; k++) result[k] = 'z';
            int i = 6; int count = 0;
            while (n > 0) {
                int rem = n % 26;
                temp[i] = (char) ((int) 'A' + rem);
                n = n / 26;
                i--; count++;
                }
            for(int k = 0; k < count; k++) result[k] = temp[++i];
            return(new String(result));
            }

        public String toString() {
            String result = "unique1 = " + unique1 + "\t";
            result += "unique2 = " + unique2 + "\t";
          result += "stringu1 = " + stringu1 + "\t";
          result += " num = " + num;
            return result;
            }

      public void addNumber(int num) {
          this.num+=num;
   }
}
```

A.1.2. Java code for ItemKeeper

```
import java.util.*;

public class ItemKeeper implements java.io.Serializable{
   private final List myItems = new ArrayList();

  void keep(Item nextItem) {
   myItems.add(nextItem);
  }
  List itemList1() { return myItems; }
}
```

A.1.3. Java code for MakeItem

```
public class MakeItems {
   Item[] items;
   static long generator, prime;

   public MakeItems(int n){
      items = new Item[n];
      if(n <= 0 || n > 100000000) n = 10;
         if(n <= 1000) {generator = 26; prime = 1009;}
         else if (n <= 10000) {generator = 59; prime = 10007;}
         else if (n <= 100000) {generator = 242; prime = 100003;}
         else if (n <= 1000000) {generator = 568; prime = 1000003;}
         else if (n <= 10000000) {generator = 1792; prime = 10000019;}
         else if (n <= 100000000) {generator = 5649; prime = 100000007;}
         else {generator = 16807; prime = 2147483647; }
         long seed = generator;
         for(int k = 0; k < n; k++) {
         seed = rand(seed, (long) n);
            items[k] = new Item((int) seed - 1, k);
                        }
         }

         private static long rand(long seed, long limit) {
         /*****************************************/
                  do {
                           seed = (generator*seed) % prime;
                           } while (seed > limit);
                  return(seed);
                  }

      public Item[] itemList(){
         return items;
      }
   }
```

A.2. Java code for storing objects in the databases

A.2.1. Db4o database

A.2.1.1. Storing objects without indexing enabled

```
// This database is created without indexing implemented

import java.util.*;
import com.db4o.Db4o;
import com.db4o.ObjectContainer;
import com.db4o.query.*;
import java.io.*;

public class CreateNoIndex {
   public static void main(String[] args) throws Exception {
      ObjectContainer db = null;

      try {
            System.out.println("Creating.....");
            FileWriter fw = new FileWriter("D:\\jim\\create-db4o-no-index.txt");
            BufferedWriter bw = new BufferedWriter (fw);
            PrintWriter p = new PrintWriter(bw);
            int limit = 100000;
               for (int size = 5000; size<=limit; size+=5000) {
                  long sum = 0;
                  long sumSquares = 0;
                  long Time = 0;
                  for (int m = 0; m<10; m++) {
                     String dbname = ("db4o-no-index-"+size+"-"+m);
                     db = Db4o.openFile(dbname);
                     Item[] items = new MakeItems(size).itemList();
                     long t1 = System.currentTimeMillis();
                        for(int k = 0; k<size; k++){
                           db.set(items[k]);
                        }
                  long t2 = System.currentTimeMillis();
                  p.println("Size = " + size + "\t" + "m = " + m + "\t" + "time = " + (t2-t1));
                  Time = t2-t1;
                  sum += Time;
                  sumSquares+= Time*Time;
```

```
                db.close();
                }
            double variance = ((double) (sumSquares - sum*sum/10) /10);
            double SD = Math.sqrt(variance);
            p.println("Average time for size " + size + "\t" + (sum /10));
            p.println("SD = " + SD);
            }
        p.close();
        bw.close();
        fw.close();
    }
    catch(Exception ex) {
        System.out.println(ex);
      }
    finally {
      db.close();
    }

  }
}
```

A.2.2.2. Storing objects with indexing enabled

```
// This database is created with indexing enabled

import java.util.*;
import com.db4o.Db4o;
import com.db4o.ObjectContainer;
import com.db4o.query.*;
import java.io.*;

public class CreateWithIndex {
   public static void main (String[] args) {
      ObjectContainer db = null;

       try {
            System.out.println("Creating.....");
            FileWriter fw = new FileWriter("D:\\jim\\create-db4o-index-4fields.txt");
            BufferedWriter bw = new BufferedWriter (fw);
            PrintWriter p = new PrintWriter(bw);
            int limit = 100000;
            for (int size = 5000; size <= limit; size+=5000) {
               long sum = 0;
               long sumSquares = 0;
```

```
                for(int m = 0; m < 10; m++) {
                   Db4o.configure().objectClass(Item.class).objectField("unique1").indexed(true);
                   Db4o.configure().objectClass(Item.class).objectField("unique2").indexed(true);
                   Db4o.configure().objectClass(Item.class).objectField("stringu2").indexed(true);
                   Db4o.configure().objectClass(Item.class).objectField("string4").indexed(true);
                   String dbname = "db4o-" + size + "-" + m;
                   db = Db4o.openFile(dbname);
                      Item[] items = new MakeItems(size).itemList();
                      long t1 = System.currentTimeMillis();
                         for(int k = 0; k<size; k++){
                            db.set(items[k].unique1);
                            db.set(items[k].unique2);
                            db.set(items[k].stringu1);
                            db.set(items[k].stringu2);
                         }
                   long t2 = System.currentTimeMillis();
                   p.println("Size = " + size + "\t" + "m = " + m + "\t" + "time = " + (t2-t1));
                   long Time = t2-t1;
                   System.out.println(Time);
                   sum += Time;
                   sumSquares+= Time*Time;
                   db.close();
                   }
                System.out.println("The time taken to store "+size+" objects is "+(sum/10));
                double variance = ((double) (sumSquares - sum*sum/10) /10);
                double SD = Math.sqrt(variance);
                p.println("Average time for size " + size + "\t" + (sum /10));
                p.println("SD = " + SD);
                System.out.println("\t" + "Standard deviation = " + SD);
                }
             p.close();
             bw.close();
             fw.close();
       }
        catch(Exception ex) {
             System.out.println(ex);
          }
       finally {
          db.close();
       }
    }
}
```

A.2.2. Neodatis database

A.2.2.1. Storing objects with indexing enabled

```
// This database is created without indexing implemented

import org.neodatis.odb.main.ODB;
import org.neodatis.odb.main.ODBFactory;
import java.util.*;
import java.io.*;

public class CreateNoIndex {

   public static String ODB_NAME = null;

   public static void main (String[] args) {
      Item[] list;
      ODB odb = null;
      try {
         System.out.println("Creating.......");
            FileWriter fw = new FileWriter("D:\\jim\\create-neodatis-no-index.txt");
            BufferedWriter bw = new BufferedWriter (fw);
            PrintWriter p = new PrintWriter(bw);
            int limit = 100000;
            for(int size = 5000; size<=limit; size+=5000) {
               long sum = 0;
               long sumSquares = 0;
               long Time = 0;
               for(int m = 0; m<10; m++) {
                  ODB_NAME = ("neodatis-no-index-"+size+"-"+m);
                  odb = ODBFactory.open(ODB_NAME);
                  list = new MakeItems(size).itemList();
                  long t1 = System.currentTimeMillis();
                     for (int k = 0; k<size; k++) {
                        odb.store(list[k]);
                     }
                  long t2 = System.currentTimeMillis();
                  p.println("Size = " + size + "\t" + "m = " + m + "\t" + "time = " + (t2-t1));
                  Time = t2-t1;
                  sum += Time;
                  sumSquares += Time*Time;
                  odb.close();
               }
               double variance = ((double) (sumSquares - sum*sum/10)) / 10;
```

```
                double SD = Math.sqrt(variance);
                p.println("Average time for size " + size + "\t" + (sum /10));
                p.println("SD = " + SD);
                System.out.println(size + " created");
                }
            p.close();
            bw.close();
            fw.close();
            }
        catch(Exception ex) {
            System.out.println(ex);

        }

    }
}
```

A.2.2.2. Storing objects with indexing enabled

```
// This database is created with indexing enabled

import org.neodatis.odb.main.ODB;
import org.neodatis.odb.main.ODBFactory;
import java.util.*;
import java.io.*;

public class CreateWithIndex {
    public static String ODB_NAME = null;

    public static void main (String[] args) {
        Item[] list;
        ODB odb = null;
        try {
            System.out.println("Creating.......");
                FileWriter fw = new FileWriter("D:\\jim\\create-neodatis-index.txt");
                BufferedWriter bw = new BufferedWriter (fw);
                PrintWriter p = new PrintWriter(bw);
                int limit = 100000;
                for(int size = 5000; size<=limit; size+=5000) {
                    long sum = 0;
                    long sumSquares = 0;
                    long Time = 0;
                    for(int m = 0; m<10; m++) {
                        ODB_NAME = ("neodatis-"+size+"-"+m);
                        odb = ODBFactory.open(ODB_NAME);
```

```
                String [] fieldNames4 = {"unique1", "unique2", "stringu1", "stringu2"};
                odb.getClassRepresentation(Item.class).addUniqueIndexOn("item-
index",fieldNames4,true);
                list = new MakeItems(size).itemList();
                long t1 = System.currentTimeMillis();
                   for (int k = 0; k<size; k++) {
                      odb.store(list[k]);
                   }
                long t2 = System.currentTimeMillis();
                p.println("Size = " + size + "\t" + "m = " + m + "\t" + "time = " + (t2-t1));
                Time = t2-t1;
                sum += Time;
                sumSquares += Time*Time;
                odb.close();
             }
             double variance = ((double) (sumSquares - sum*sum/10)) / 10;
             double SD = Math.sqrt(variance);
             p.println("Average time for size " + size + "\t" + (sum /10));
             p.println("SD = " + SD);
      }
            p.close();
            bw.close();
            fw.close();
            }
      catch(Exception ex) {
            System.out.println(ex);

      }

   }

}
```

A.2.3. Perst database

A.2.3.1. Storing objects without indexing enabled

```
// This database is created without indexing implemented

import org.garret.perst.*;
import org.garret.perst.Storage;
import java.util.*;
import org.garret.perst.StorageFactory;
import org.garret.perst.Storage;
import java.io.*;
```

```
public class CreateNoIndex {
      public static void main (String[] args) {
         try {
            System.out.println("Creating .......");
            FileWriter fw = new FileWriter("D:\\jim\\create-perst-no-index.txt");
            BufferedWriter bw = new BufferedWriter (fw);
            PrintWriter p = new PrintWriter(bw);
            int limit = 100000;
            for (int size = 5000; size <= limit; size+=5000) {
               long sum = 0;
               long sumSquares = 0;
               for(int m = 0; m < 10; m++) {
                  String dbname = "perst-no-index-" +size+ "-" + m;
                  Storage db = StorageFactory.getInstance().createStorage();
                  db.open(dbname,Storage.INFINITE_PAGE_POOL);
                  IPersistentList root = (IPersistentList) db.getRoot();
                  if(root == null) {
                     root = db.createList();
                     db.setRoot(root);
                  }  // if root
               Item[] items = new MakeItems(size).itemList();
               long t1 = System.currentTimeMillis();
               for (int k = 0;k<size;k++) {
                     root.add(items[k]);
                     }   // for int k
               long t2 = System.currentTimeMillis();
               p.println("Size = " + size + "\t" + "m = " + m + "\t" + "time = " + (t2-t1));
               sum += (t2-t1);
               sumSquares+= (t2-t1)*(t2-t1);
               db.close();
               System.out.println("Created size = " + size + " m = " + m);
               }  // for int m
               double variance = ((double) (sumSquares - sum*sum/10) /10);
               double SD = Math.sqrt(variance);
               p.println("Average time for size " + size + "\t" + (((double)sum) /10));
               p.println("SD = " + SD);
               } // for int size
               p.close();
               bw.close();
               fw.close();
         }  // try
         catch(Exception ex) {
            System.out.println(ex);
         }
      }
}
```

A.2.3.2. Storing objects with indexing enabled

```
// This database is created with indexing enabled

import org.garret.perst.*;
import org.garret.perst.Storage;
import org.garret.perst.Index;
import java.util.*;
import org.garret.perst.StorageFactory;
import org.garret.perst.Storage;
import org.garret.perst.Key;
import org.garret.perst.Index;
import java.io.*;

class Indices extends Persistent {
            Index unique1Index;
            Index unique2Index;
            Index stringu1Index;
            Index stringu2Index;

            public Indices() {}
}

public class CreateWithIndex4 {
      public static void main (String[] args) {

         try {
            System.out.println("Creating…….");
            FileWriter fw = new FileWriter("D:\\jim\\create-perst-index-4fields.txt");
            BufferedWriter bw = new BufferedWriter (fw);
            PrintWriter p = new PrintWriter(bw);
            int limit = 100000;
            for (int size = 5000; size <= limit; size+=5000) {
               long sum = 0;
               long sumSquares = 0;
               for(int m = 0; m < 10; m++) {
                  String dbname = "perst-" + size + "-" + m;
                  Storage db = StorageFactory.getInstance().createStorage();
                  db.open(dbname,Storage.INFINITE_PAGE_POOL);
                  Indices root = (Indices) db.getRoot();
                  if(root == null) {
                     root = new Indices();
                     root.unique1Index = db.createIndex(int.class, true);
                     root.unique2Index = db.createIndex(int.class, true);
                     root.stringu1Index = db.createIndex(String.class, true);
                     root.stringu2Index = db.createIndex(String.class, true);
```

```
                db.setRoot(root);
            }  // if root

            Item[] items = new MakeItems(size).itemList();
            long t1 = System.currentTimeMillis();
            for (int k = 0;k<size;k++) {
               root.unique1Index.put(new Key(items[k].unique1),items[k]);
               root.unique2Index.put(new Key(items[k].unique2),items[k]);
               root.stringu1Index.put(new Key(items[k].stringu1),items[k]);
               root.stringu2Index.put(new Key(items[k].stringu2),items[k]);
            }   // for int k
            long t2 = System.currentTimeMillis();
            p.println("Size = " + size + "\t" + "m = " + m + "\t" + "time = " + (t2-t1));
            sum += (t2-t1);
            sumSquares+= (t2-t1)*(t2-t1);
            db.close();
            System.out.println("Created size = " + size + " m = " + m);
            }  // for int m
            double variance = ((double) (sumSquares - sum*sum/10) /10);
            double SD = Math.sqrt(variance);
            p.println("Average time for size " + size + "\t" + (sum /10));
            p.println("SD = " + SD);
         } // for int size
            p.close();
            bw.close();
            fw.close();
      }  // try
      catch(Exception ex) {
         System.out.println(ex);
      }

   }
}
```

A.2.4. Prevayler database

A.2.4.1. Storing objects without indexing enabled

```
// This database is created without indexing implemented

import org.prevayler.Prevayler;
import org.prevayler.PrevaylerFactory;
import java.util.*;
import java.io.*;
```

```
public class CreateNoIndex {

   public static void main(String[] args) throws Exception {

      Prevayler prevayler = null;
      ItemKeeper itemKeeper = null;

         System.out.println("Creating .......");
            FileWriter fw = new FileWriter("D:\\jim\\create-prevayler-no-index.txt");
            BufferedWriter bw = new BufferedWriter (fw);
            PrintWriter p = new PrintWriter(bw);
            int limit = 100000;
            for(int size = 5000; size<=limit; size+=5000) {
                  long Time = 0;
                  String dbname = ("prevayler-no-index-"+size);
                  prevayler = PrevaylerFactory.createPrevayler(new ItemKeeper(), dbname);
                  itemKeeper = (ItemKeeper)prevayler.prevalentSystem();
                  Item[] items = new MakeItems(size).itemList();
                  long t1 = System.currentTimeMillis();
                     for (int k = 0; k<size; k++) {
                        prevayler.execute(new InsertItem(items[k]));
                     }
                  long t2 = System.currentTimeMillis();
                  p.println("Size = " + size + "\t" + "time = " + (t2-t1));
                  Time = t2-t1;
                  System.out.println(Time);

            }  // for ssize
         p.close();
         bw.close();
         fw.close();
         } // main
}
```

A.3. Java code for searching for objects in the databases

A.3.1. Without indexing enabled

A.3.1.1. Db4o database

// Searching for the field *unique1* without indexing

```
import java.util.*;
import com.db4o.Db4o;
import com.db4o.ObjectContainer;
import com.db4o.ObjectSet;
import com.db4o.query.*;
import java.io.*;

public class SearchNoIndexUnique1 {
   public static void main (String[] args) {

      ObjectContainer db = null;

         try {
            System.out.println("Searching .....");
            FileWriter fw = new FileWriter("D:\\jim\\search-db4o-no-index-unique1.txt");
            BufferedWriter bw = new BufferedWriter (fw);
            PrintWriter p = new PrintWriter(bw);
            int limit = 100000;
            for (int size = 5000; size <= limit; size+=5000) {
               long sum = 0;
               long sumSquares = 0;
               for (int m = 0; m<10; m++) {
                  long t1 = System.currentTimeMillis();
                  String dbname = ("db4o-no-index-"+size+"-"+m);
                  db = Db4o.openFile(dbname);
                  Query query = db.query();
                  query.constrain(Item.class);
                  query.descend("unique1").constrain(new Integer(300));
                  ObjectSet result = query.execute();
                  long t2 = System.currentTimeMillis();
                  sum += (t2-t1);
                  sumSquares+= (t2-t1)*(t2-t1);
                  db.close();
                  }
               System.out.println("Searched size = " + size + " For unique1 = 300");
               double variance = ((double) (sumSquares - sum*sum/10) /10);
               double SD = Math.sqrt(variance);
               p.println("Searching time for size " + size + "\t" + (((double)sum) /10));
               p.println("SD = " + SD);
            }
            p.close();
            bw.close();
            fw.close();
         }
         catch(Exception ex) {
            System.out.println(ex);
```

```
            ex.printStackTrace();
        }
    }
}


// Searching for the field unique2 without indexing

import java.util.*;
import com.db4o.Db4o;
import com.db4o.ObjectContainer;
import com.db4o.ObjectSet;
import com.db4o.query.*;
import java.io.*;

public class SearchNoIndexUnique2 {
    public static void main (String[] args) {

        ObjectContainer db = null;

            try {
                System.out.println("Searching .....");
                FileWriter fw = new FileWriter("D:\\jim\\search-db4o-no-index-unique2.txt");
                BufferedWriter bw = new BufferedWriter (fw);
                PrintWriter p = new PrintWriter(bw);
                int limit = 100000;
                for (int size = 5000; size <= limit; size+=5000) {
                    long sum = 0;
                    long sumSquares = 0;
                    for (int m = 0; m<10; m++) {
                        long t1 = System.currentTimeMillis();
                        String dbname = ("db4o-no-index-"+size+"-"+m);
                        db = Db4o.openFile(dbname);
                        Query query = db.query();
                        query.constrain(Item.class);
                        query.descend("unique2").constrain(new Integer(300));
                        ObjectSet result = query.execute();
                        long t2 = System.currentTimeMillis();
                        sum += (t2-t1);
                        sumSquares+= (t2-t1)*(t2-t1);
                        db.close();
                        }
                    System.out.println("Searched size = " + size + " For unique2 = 300");
                    double variance = ((double) (sumSquares - sum*sum/10) /10);
                    double SD = Math.sqrt(variance);
                    p.println("Searching time for size " + size + "\t" + (((double)sum) /10));
```

```
                p.println("SD = " + SD);
            }
            p.close();
            bw.close();
            fw.close();
        }
        catch(Exception ex) {
            System.out.println(ex);
            ex.printStackTrace();
        }
    }

}


// Searching for the field stringu1without indexing

import java.util.*;
import com.db4o.Db4o;
import com.db4o.ObjectContainer;
import com.db4o.ObjectSet;
import com.db4o.query.*;
import java.io.*;

public class SearchNoIndexStringu1 {
    public static void main (String[] args) {

        String x45 = "xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx";
        ObjectContainer db = null;

            try {
                System.out.println("Searching .....");
                FileWriter fw = new FileWriter("D:\\jim\\search-db4o-no-index-stringu1.txt");
                BufferedWriter bw = new BufferedWriter (fw);
                PrintWriter p = new PrintWriter(bw);
                int limit = 100000;
                for (int size = 5000; size <= limit; size+=5000) {
                    long sum = 0;
                    long sumSquares = 0;
                    for (int m = 0; m<10; m++) {
                        long t1 = System.currentTimeMillis();
                        String dbname = ("db4o-no-index-"+size+"-"+m);
                        db = Db4o.openFile(dbname);
                        Query query = db.query();
                        query.constrain(Item.class);
                        query.descend("stringu1").constrain(new String(convert(300)+ x45));
```

```
                ObjectSet result = query.execute();
                long t2 = System.currentTimeMillis();
                while(result.hasNext()) {
                        System.out.println(result.next());
                                }
                db.close();
                sum += (t2-t1);
                sumSquares+= (t2-t1)*(t2-t1);
                //System.out.println(result.size());
                }
              System.out.println("Searched size = " + size + " For stringu1 = convert(300)+
x45");
              double variance = ((double) (sumSquares - sum*sum/10) /10);
              double SD = Math.sqrt(variance);
              p.println("Searching time for size " + size + "\t" + (((double)sum) /10));
              p.println("SD = " + SD);
            }
            p.close();
            bw.close();
            fw.close();
        }
        catch(Exception ex) {
            System.out.println(ex);
            ex.printStackTrace();
        }
    }
    private  static String convert(int n) {
        /*****************************/
            char[] result = new char[7];
                char[] temp = new char[7];
                for(int k = 0; k < 7; k++) result[k] = 'z';
                int i = 6; int count = 0;
                while (n > 0) {
                        int rem = n % 26;
                        temp[i] = (char) ((int) 'A' + rem);
                        n = n / 26;
                        i--; count++;
                        }
                for(int k = 0; k < count; k++) result[k] = temp[++i];
                return(new String(result));
                }

}
```

// Searching for the field *stringu2*without indexing

```
import java.util.*;
import com.db4o.Db4o;
import com.db4o.ObjectContainer;
import com.db4o.ObjectSet;
import com.db4o.query.*;
import java.io.*;

public class SearchNoIndexStringu2 {
   public static void main (String[] args) {

      String x45 = "xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx";
      ObjectContainer db = null;

         try {
            System.out.println("Searching .....");
            FileWriter fw = new FileWriter("D:\\jim\\search-db4o-no-index-stringu2.txt");
            BufferedWriter bw = new BufferedWriter (fw);
            PrintWriter p = new PrintWriter(bw);
            int limit = 100000;
            for (int size = 5000; size <= limit; size+=5000) {
               long sum = 0;
               long sumSquares = 0;
               for (int m = 0; m<10; m++) {
                  long t1 = System.currentTimeMillis();
                  String dbname = ("db4o-no-index-"+size+"-"+m);
                  db = Db4o.openFile(dbname);
                  Query query = db.query();
                  query.constrain(Item.class);
                  query.descend("stringu2").constrain(new String(convert(300)+ x45));
                  ObjectSet result = query.execute();
                  long t2 = System.currentTimeMillis();
                  while(result.hasNext()) {
                        System.out.println(result.next());
                              }
                  db.close();
                  sum += (t2-t1);
                  sumSquares+= (t2-t1)*(t2-t1);
                  //System.out.println(result.size());
                  }
               System.out.println("Searched size = " + size + " For stringu2 = convert(300)+
x45");
               double variance = ((double) (sumSquares - sum*sum/10) /10);
               double SD = Math.sqrt(variance);
               p.println("Searching time for size " + size + "\t" + (((double)sum) /10));
```

```
                p.println("SD = " + SD);
            }
            p.close();
            bw.close();
            fw.close();
        }
        catch(Exception ex) {
            System.out.println(ex);
            ex.printStackTrace();
        }
    }
    private  static String convert(int n) {
        /*****************************/
            char[] result = new char[7];
                char[] temp = new char[7];
                for(int k = 0; k < 7; k++) result[k] = 'z';
                int i = 6; int count = 0;
                while (n > 0) {
                        int rem = n % 26;
                        temp[i] = (char) ((int) 'A' + rem);
                        n = n / 26;
                        i--; count++;
                        }
                for(int k = 0; k < count; k++) result[k] = temp[++i];
                return(new String(result));
     }
}
```

// Searching for the field *string4*without indexing

```
import java.util.*;
import com.db4o.Db4o;
import com.db4o.ObjectContainer;
import com.db4o.ObjectSet;
import com.db4o.query.*;
import java.io.*;

public class SearchNoIndexString4 {
    public static void main (String[] args) {
        String x45 = "xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx";
        ObjectContainer db = null;
            try {
                System.out.println("Searching .....");
                FileWriter fw = new FileWriter("D:\\jim\\search-db4o-no-index-string4.txt");
```

```
            BufferedWriter bw = new BufferedWriter (fw);
            PrintWriter p = new PrintWriter(bw);
            int limit = 100000;
            for (int size = 5000; size <= limit; size+=5000) {
               long sum = 0;
               long sumSquares = 0;
               for (int m = 0; m<10; m++) {
                  long t1 = System.currentTimeMillis();
                  String dbname = ("db4o-no-index-"+size+"-"+m);
                  db = Db4o.openFile(dbname);
                  Query query = db.query();
                  query.constrain(Item.class);
                  query.descend("string4").constrain(new String("AAAA"+x45+ "xxx"));
                  ObjectSet result = query.execute();
                  long t2 = System.currentTimeMillis();
                  while(result.hasNext()) {
                        System.out.println(result.next());
                                 }
                  db.close();
                  sum += (t2-t1);
                  sumSquares+= (t2-t1)*(t2-t1);
                  }
               System.out.println("Searched size = " + size + " For string4 = 'AAAA"+x45+
"xxx'");
               double variance = ((double) (sumSquares - sum*sum/10) /10);
               double SD = Math.sqrt(variance);
               p.println("Searching time for size " + size + "\t" + (((double)sum) /10));
               p.println("SD = " + SD);
            }
            p.close();
            bw.close();
            fw.close();
         }
         catch(Exception ex) {
            System.out.println(ex);
            ex.printStackTrace();
         }
   }
   private  static String convert(int n) {
         /*****************************/
            char[] result = new char[7];
                  char[] temp = new char[7];
                  for(int k = 0; k < 7; k++) result[k] = 'z';
                  int i = 6; int count = 0;
                  while (n > 0) {
                           int rem = n % 26;
```

```
                temp[i] = (char) ((int) 'A' + rem);
                n = n / 26;
                i--; count++;
                }
            for(int k = 0; k < count; k++) result[k] = temp[++i];
            return(new String(result));
        }
}
// Searching for one percent of objects without indexing

import java.util.*;
import com.db4o.Db4o;
import com.db4o.ObjectContainer;
import com.db4o.ObjectSet;
import com.db4o.query.*;
import java.io.*;

public class SearchNoIndexOnePercent {
   public static void main (String[] args) {

      ObjectContainer db = null;
         try {
            System.out.println("Searching .....");
            FileWriter fw = new FileWriter("D:\\jim\\search-db4o-no-index-onePercent.txt");
            BufferedWriter bw = new BufferedWriter (fw);
            PrintWriter p = new PrintWriter(bw);
            int limit = 100000;
            for (int size = 5000; size <= limit; size+=5000) {
               long sum = 0;
               long sumSquares = 0;
               for (int m = 0; m<10; m++) {
                  long t1 = System.currentTimeMillis();
                  String dbname = ("db4o-no-index-"+size+"-"+m);
                  db = Db4o.openFile(dbname);
                  Query query = db.query();
                  query.constrain(Item.class);
                  query.descend("onePercent").constrain(new Integer(5));
                  ObjectSet result = query.execute();
                  long t2 = System.currentTimeMillis();
                  while(result.hasNext()) {
                        System.out.println(result.next());
                        }
                  db.close();
                  sum += (t2-t1);
                  sumSquares+= (t2-t1)*(t2-t1);
                  }
```

```
            System.out.println("Searched size = " + size + " For onePercent = 5");
            double variance = ((double) (sumSquares - sum*sum/10) /10);
            double SD = Math.sqrt(variance);
            p.println("Searching time for size " + size + "\t" + (((double)sum) /10));
            p.println("SD = " + SD);
         }
         p.close();
         bw.close();
         fw.close();
      }
      catch(Exception ex) {
         System.out.println(ex);
         ex.printStackTrace();
      }
   }
}


// Searching for ten percent of objects without indexing

import java.util.*;
import com.db4o.Db4o;
import com.db4o.ObjectContainer;
import com.db4o.ObjectSet;
import com.db4o.query.*;
import java.io.*;

public class SearchNoIndexTenPercent {
   public static void main (String[] args) {

      ObjectContainer db = null;
         try {
            System.out.println("Searching .....");
            FileWriter fw = new FileWriter("D:\\jim\\search-db4o-no-index-tenPercent.txt");
            BufferedWriter bw = new BufferedWriter (fw);
            PrintWriter p = new PrintWriter(bw);
            int limit = 100000;
            for (int size = 5000; size <= limit; size+=5000) {
               long sum = 0;
               long sumSquares = 0;
               for (int m = 0; m<10; m++) {
                  long t1 = System.currentTimeMillis();
                  String dbname = ("db4o-no-index-"+size+"-"+m);
                  db = Db4o.openFile(dbname);
                  Query query = db.query();
                  query.constrain(Item.class);
```

```
                query.descend("tenPercent").constrain(new Integer(4));
                ObjectSet result = query.execute();
                long t2 = System.currentTimeMillis();
                sum += (t2-t1);
                sumSquares+= (t2-t1)*(t2-t1);
                db.close();
                }
            System.out.println("Searched size = " + size + " For tenPercent = 4");
            double variance = ((double) (sumSquares - sum*sum/10) /10);
            double SD = Math.sqrt(variance);
            p.println("Searching time for size " + size + "\t" + (((double)sum) /10));
            p.println("SD = " + SD);
          }
          p.close();
          bw.close();
          fw.close();
        }
        catch(Exception ex) {
          System.out.println(ex);
          ex.printStackTrace();
        }
    }
}


// Searching for twenty percent of objects without indexing

import java.util.*;
import com.db4o.Db4o;
import com.db4o.ObjectContainer;
import com.db4o.ObjectSet;
import com.db4o.query.*;
import java.io.*;

public class SearchNoIndexTwentyPercent {
   public static void main (String[] args) {

      ObjectContainer db = null;

        try {
          System.out.println("Searching .....");
          FileWriter fw = new FileWriter("D:\\jim\\search-db4o-no-index-twentyPercent.txt");
          BufferedWriter bw = new BufferedWriter (fw);
          PrintWriter p = new PrintWriter(bw);
          int limit = 100000;
          for (int size = 5000; size <= limit; size+=5000) {
```

```
                long sum = 0;
                long sumSquares = 0;
                for (int m = 0; m<10; m++) {
                   long t1 = System.currentTimeMillis();
                   String dbname = ("db4o-no-index-"+size+"-"+m);
                   db = Db4o.openFile(dbname);
                   Query query = db.query();
                   query.constrain(Item.class);
                   query.descend("twentyPercent").constrain(new Integer(2));
                   ObjectSet result = query.execute();
                   long t2 = System.currentTimeMillis();
                   sum += (t2-t1);
                   sumSquares+= (t2-t1)*(t2-t1);
                   db.close();
                   }
                System.out.println("Searched size = " + size + " For twentyPercent = 2");
                double variance = ((double) (sumSquares - sum*sum/10) /10);
                double SD = Math.sqrt(variance);
                p.println("Searching time for size " + size + "\t" + (((double)sum) /10));
                p.println("SD = " + SD);
             }
             p.close();
             bw.close();
             fw.close();
          }
          catch(Exception ex) {
             System.out.println(ex);
             ex.printStackTrace();
          }
    }
}


// Searching for fifty percent of objects without indexing

import java.util.*;
import com.db4o.Db4o;
import com.db4o.ObjectContainer;
import com.db4o.ObjectSet;
import com.db4o.query.*;
import java.io.*;

public class SearchNoIndexFiftyPercent {
    public static void main (String[] args) {

        ObjectContainer db = null;
```

```
        try {
           System.out.println("Searching .....");
           FileWriter fw = new FileWriter("D:\\jim\\search-db4o-no-index-fiftyPercent.txt");
           BufferedWriter bw = new BufferedWriter (fw);
           PrintWriter p = new PrintWriter(bw);
           int limit = 100000;
           for (int size = 5000; size <= limit; size+=5000) {
              long sum = 0;
              long sumSquares = 0;
              for (int m = 0; m<10; m++) {
                 long t1 = System.currentTimeMillis();
                 String dbname = ("db4o-no-index-"+size+"-"+m);
                 db = Db4o.openFile(dbname);
                 Query query = db.query();
                 query.constrain(Item.class);
                 query.descend("fiftyPercent").constrain(new Integer(1));
                 ObjectSet result = query.execute();
                 long t2 = System.currentTimeMillis();
                 while(result.hasNext()) {
                       System.out.println(result.next());
                             }
                 db.close();
                 sum += (t2-t1);
                 sumSquares+= (t2-t1)*(t2-t1);
                 }
              System.out.println("Searched size = " + size + " For fiftyPercent = 1");
              double variance = ((double) (sumSquares - sum*sum/10) /10);
              double SD = Math.sqrt(variance);
              p.println("Searching time for size " + size + "\t" + (((double)sum) /10));
              p.println("SD = " + SD);
           }
           p.close();
           bw.close();
           fw.close();
        }
        catch(Exception ex) {
           System.out.println(ex);
           ex.printStackTrace();
        }
     }
}
```

A.3.2.2. Neodatis database

// Searching for the field *unique1* without indexing

```
import org.neodatis.odb.core.Objects;
import org.neodatis.odb.main.ODB;
import org.neodatis.odb.main.ODBFactory;
import java.util.*;
import org.neodatis.odb.core.query.IQuery;
import org.neodatis.odb.core.query.criteria.CriteriaQuery;
import org.neodatis.odb.core.query.criteria.Where;
import java.io.*;

public class SearchNoIndexUnique1 {

   public static void main(String[] args) {
      ODB odb = null;
         try {
            System.out.println("Searching .....");
            FileWriter fw = new FileWriter("D:\\jim\\search-neodatis-no-index-unique1.txt");
            BufferedWriter bw = new BufferedWriter (fw);
            PrintWriter p = new PrintWriter(bw);
            int limit = 100000;
            for (int size = 5000; size <= limit; size+=5000) {
               long sum = 0;
               long sumSquares = 0;
               int repeats = 10;
               for(int m = 0; m < repeats; m++) {
                  long t1 = System.currentTimeMillis();
                  String ODB_NAME = ("neodatis-no-index-"+size+"-"+m);
                  odb = ODBFactory.open(ODB_NAME);
                  IQuery query = new CriteriaQuery(Item.class, Where.equal("unique1", 300));
                  Objects value = odb.getObjects(query);
                  long t2 = System.currentTimeMillis();
                  System.out.println("Items selected: "+value.size());
                  int i = 1;
                     while (value.hasNext()) {
                        System.out.println((i++)+ "\t:" + value.next()+ "\t");
                     }
                  odb.close();
               sum += (t2-t1);
               sumSquares+= (t2-t1)*(t2-t1);
               } //for int m
            System.out.println("Searched size = " + size + " For unique1 = 300");
            double variance = ((double) (sumSquares - sum*sum/repeats) /repeats);
            double SD = Math.sqrt(variance);
```

```
            p.println("Searching time for size " + size + "\t" + (((double)sum) /repeats));
            p.println("SD = " + SD);
            System.out.println("The time taken to select unique1 is "+(((double)sum) /repeats));
         }
            p.close();
            bw.close();
            fw.close();
           }
      catch(Exception ex) {
            System.out.println(ex);
            ex.printStackTrace();
            //odb.close();
      }
   }
}
```

// Searching for the field *unique2* without indexing

```
import org.neodatis.odb.core.Objects;
import org.neodatis.odb.main.ODB;
import org.neodatis.odb.main.ODBFactory;
import java.util.*;
import org.neodatis.odb.core.query.IQuery;
import org.neodatis.odb.core.query.criteria.CriteriaQuery;
import org.neodatis.odb.core.query.criteria.Where;
import java.io.*;

public class SearchNoIndexUnique2 {

   public static void main(String[] args) {
      ODB odb = null;
         try {
            System.out.println("Searching ........");
            FileWriter fw = new FileWriter("D:\\jim\\search-neodatis-no-index-unique2.txt");
            BufferedWriter bw = new BufferedWriter (fw);
            PrintWriter p = new PrintWriter(bw);
            int limit = 100000;
            for (int size = 5000; size <= limit; size+=5000) {
               long sum = 0;
               long sumSquares = 0;
               int repeats = 10;
               for(int m = 0; m < repeats; m++) {
                  long t1 = System.currentTimeMillis();
                  String ODB_NAME = ("neodatis-no-index-"+size+"-"+m);
                  odb = ODBFactory.open(ODB_NAME);
                  IQuery query = new CriteriaQuery(Item.class, Where.equal("unique2", 300));
```

```
                Objects value = odb.getObjects(query);
                long t2 = System.currentTimeMillis();
                System.out.println("Items selected: "+value.size());
                int i = 1;
                while (value.hasNext()) {
                System.out.println((i++)+ "\t:" + value.next()+ "\t");
                }
                odb.close();
              sum += (t2-t1);
              sumSquares+= (t2-t1)*(t2-t1);
              } //for int m
            System.out.println("Searched size = " + size + " For unique2 = 300");
            double variance = ((double) (sumSquares - sum*sum/repeats) /repeats);
            double SD = Math.sqrt(variance);
            p.println("Searching time for size " + size + "\t" + (((double)sum) /repeats));
            p.println("SD = " + SD);
            System.out.println("The time taken to select unique1 is "+(((double)sum) /repeats));
          }
            p.close();
            bw.close();
            fw.close();
           }
       catch(Exception ex) {
            System.out.println(ex);
            ex.printStackTrace();
       }
    }
}


// Searching for the field stringu1without indexing

import org.neodatis.odb.core.Objects;
import org.neodatis.odb.main.ODB;
import org.neodatis.odb.main.ODBFactory;
import java.util.*;
import org.neodatis.odb.core.query.IQuery;
import org.neodatis.odb.core.query.criteria.CriteriaQuery;
import org.neodatis.odb.core.query.criteria.Where;
import java.io.*;

public class SearchNoIndexStringu1 {

    //public static String ODB_NAME = null;

    public static void main(String[] args) {
```

```
    String x45 = "xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx";
    ODB odb = null;
      try {
        System.out.println("Searching .....");
        FileWriter fw = new FileWriter("D:\\jim\\search-neodatis-no-index-stringu1.txt");
        BufferedWriter bw = new BufferedWriter (fw);
        PrintWriter p = new PrintWriter(bw);
        int limit = 100000;
        for (int size = 5000; size <= limit; size+=5000) {
          long sum = 0;
          long sumSquares = 0;
          int repeats = 10;
          for(int m = 0; m < repeats; m++) {
            long t1 = System.currentTimeMillis();
            String ODB_NAME = ("neodatis-no-index-"+size+"-"+m);
            odb = ODBFactory.open(ODB_NAME);
            IQuery query = new CriteriaQuery(Item.class, Where.equal("stringu1",
convert(300)+ x45 ));
            Objects value = odb.getObjects(query);
            long t2 = System.currentTimeMillis();
            System.out.println("Items selected: "+value.size());
            int i = 1;
            while (value.hasNext()) {
            System.out.println((i++)+ "\t:" + value.next()+ "\t");
            }
            odb.close();
          sum += (t2-t1);
          sumSquares+= (t2-t1)*(t2-t1);
          } //for int m
        System.out.println("Searched size = " + size + " For stringu1 = convert(300)+ x45");
        double variance = ((double) (sumSquares - sum*sum/repeats) /repeats);
        double SD = Math.sqrt(variance);
        p.println("Searching time for size " + size + "\t" + (((double)sum) /repeats));
        p.println("SD = " + SD);
        System.out.println("The time taken to select unique1 is "+(((double)sum) /repeats));
      }
        p.close();
        bw.close();
        fw.close();
       }
    catch(Exception ex) {
        System.out.println(ex);
        ex.printStackTrace();
    }
  }
```

```
private static String convert(int n) {
        /*****************************/
                char[] result = new char[7];
                char[] temp = new char[7];
                for(int k = 0; k < 7; k++) result[k] = 'z';
                int i = 6; int count = 0;
                while (n > 0) {
                        int rem = n % 26;
                        temp[i] = (char) ((int) 'A' + rem);
                        n = n / 26;
                        i--; count++;
                        }
                for(int k = 0; k < count; k++) result[k] = temp[++i];
                return(new String(result));
        }
}
```

// Searching for the field *stringu2*without indexing

```
import org.neodatis.odb.core.Objects;
import org.neodatis.odb.main.ODB;
import org.neodatis.odb.main.ODBFactory;
import java.util.*;
import org.neodatis.odb.core.query.IQuery;
import org.neodatis.odb.core.query.criteria.CriteriaQuery;
import org.neodatis.odb.core.query.criteria.Where;
import java.io.*;

public class SearchNoIndexStringu2 {

   public static void main(String[] args) {
      String x45 = "xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx";
      ODB odb = null;
         try {
            System.out.println("Searching .....");
            FileWriter fw = new FileWriter("D:\\jim\\search-neodatis-no-index-stringu2.txt");
            BufferedWriter bw = new BufferedWriter (fw);
            PrintWriter p = new PrintWriter(bw);
            int limit = 100000;
            for (int size = 5000; size <= limit; size+=5000) {
               long sum = 0;
               long sumSquares = 0;
               int repeats = 10;
               for(int m = 0; m < repeats; m++) {
                  long t1 = System.currentTimeMillis();
                  String ODB_NAME = ("neodatis-no-index-"+size+"-"+m);
```

```
                odb = ODBFactory.open(ODB_NAME);
                IQuery query = new CriteriaQuery(Item.class, Where.equal("stringu2",
convert(300)+ x45 ));
                Objects value = odb.getObjects(query);
                long t2 = System.currentTimeMillis();
                System.out.println("Items selected: "+value.size());
                int i = 1;
                while (value.hasNext()) {
                System.out.println((i++)+ "\t:" + value.next()+ "\t");
                }
                odb.close();
              sum += (t2-t1);
              sumSquares+= (t2-t1)*(t2-t1);
              } //for int m
            System.out.println("Searched size = " + size + " For stringu2 = convert(300)+ x45");
            double variance = ((double) (sumSquares - sum*sum/repeats) /repeats);
            double SD = Math.sqrt(variance);
            p.println("Searching time for size " + size + "\t" + (((double)sum) /repeats));
            p.println("SD = " + SD);
            System.out.println("The time taken to select unique1 is "+(((double)sum) /repeats));
        }
            p.close();
            bw.close();
            fw.close();
           }
      catch(Exception ex) {
            System.out.println(ex);
            ex.printStackTrace();
      }
   }
    private static String convert(int n) {
        /*****************************/
                char[] result = new char[7];
                char[] temp = new char[7];
                for(int k = 0; k < 7; k++) result[k] = 'z';
                int i = 6; int count = 0;
                while (n > 0) {
                        int rem = n % 26;
                        temp[i] = (char) ((int) 'A' + rem);
                        n = n / 26;
                        i--; count++;
                        }
                for(int k = 0; k < count; k++) result[k] = temp[++i];
                return(new String(result));
        }
}
```

// Searching for the field *string4*without indexing

```
import org.neodatis.odb.core.Objects;
import org.neodatis.odb.main.ODB;
import org.neodatis.odb.main.ODBFactory;
import java.util.*;
import org.neodatis.odb.core.query.IQuery;
import org.neodatis.odb.core.query.criteria.CriteriaQuery;
import org.neodatis.odb.core.query.criteria.Where;
import java.io.*;

public class SearchNoIndexString4 {

   public static void main(String[] args) {
      String x45 = "xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx";
      ODB odb = null;
         try {
            System.out.println("Searching .....");
            FileWriter fw = new FileWriter("D:\\jim\\search-neodatis-no-index-string4.txt");
            BufferedWriter bw = new BufferedWriter (fw);
            PrintWriter p = new PrintWriter(bw);
            int limit = 100000;
            for (int size = 5000; size <= limit; size+=5000) {
               long sum = 0;
               long sumSquares = 0;
               int repeats = 10;
               for(int m = 0; m < repeats; m++) {
                  long t1 = System.currentTimeMillis();
                  String ODB_NAME = ("neodatis-no-index-"+size+"-"+m);
                  odb = ODBFactory.open(ODB_NAME);
                  IQuery query = new CriteriaQuery(Item.class, Where.equal("string4",
"AAAA"+x45+ "xxx" ));
                  Objects value = odb.getObjects(query);
                  long t2 = System.currentTimeMillis();
                  System.out.println("Items selected: "+value.size());
                  int i = 1;
                  while (value.hasNext()) {
                  System.out.println((i++)+ "\t:" + value.next()+ "\t");
                  }
                  odb.close();
               sum += (t2-t1);
               sumSquares+= (t2-t1)*(t2-t1);
               } //for int m
            System.out.println("Searched size = " + size + " For string4 = AAAA +x45+ xxx");
            double variance = ((double) (sumSquares - sum*sum/repeats) /repeats);
            double SD = Math.sqrt(variance);
```

```
            p.println("Searching time for size " + size + "\t" + (((double)sum) /repeats));
            p.println("SD = " + SD);
            System.out.println("The time taken to select unique1 is "+(((double)sum) /repeats));
        }
            p.close();
            bw.close();
            fw.close();
           }
      catch(Exception ex) {
            System.out.println(ex);
            ex.printStackTrace();
      }
   }


private static String convert(int n) {
        /*****************************/
                char[] result = new char[7];
                char[] temp = new char[7];
                for(int k = 0; k < 7; k++) result[k] = 'z';
                int i = 6; int count = 0;
                while (n > 0) {
                        int rem = n % 26;
                        temp[i] = (char) ((int) 'A' + rem);
                        n = n / 26;
                        i--; count++;
                        }
                for(int k = 0; k < count; k++) result[k] = temp[++i];
                return(new String(result));
        }
}


// Searching for one percent of objects without indexing

import org.neodatis.odb.core.Objects;
import org.neodatis.odb.main.ODB;
import org.neodatis.odb.main.ODBFactory;
import java.util.*;
import org.neodatis.odb.core.query.IQuery;
import org.neodatis.odb.core.query.criteria.CriteriaQuery;
import org.neodatis.odb.core.query.criteria.Where;
import java.io.*;

public class SearchNoIndexOnePercent {
   public static void main(String[] args) {
      ODB odb = null;
```

```
        try {
            System.out.println("Searching .....");
            FileWriter fw = new FileWriter("D:\\jim\\search-neodatis-no-index-onePercent.txt");
            BufferedWriter bw = new BufferedWriter (fw);
            PrintWriter p = new PrintWriter(bw);
            int limit = 100000;
            for (int size = 5000; size <= limit; size+=5000) {
                long sum = 0;
                long sumSquares = 0;
                int repeats = 10;
                for(int m = 0; m < repeats; m++) {
                    long t1 = System.currentTimeMillis();
                    String ODB_NAME = ("neodatis-no-index-"+size+"-"+m);
                    odb = ODBFactory.open(ODB_NAME);
                    IQuery query = new CriteriaQuery(Item.class, Where.equal("onePercent", 5));
                    Objects value = odb.getObjects(query);
                    long t2 = System.currentTimeMillis();
                    System.out.println("Items selected: "+value.size());
                    int i = 1;
                    while (value.hasNext()) {
                    System.out.println((i++)+ "\t:" + value.next()+ "\t");
                    }
                    odb.close();
                sum += (t2-t1);
                sumSquares+= (t2-t1)*(t2-t1);
                } //for int m
            System.out.println("Searched size = " + size + " For onePercent = 5");
            double variance = ((double) (sumSquares - sum*sum/repeats) /repeats);
            double SD = Math.sqrt(variance);
            p.println("Searching time for size " + size + "\t" + (((double)sum) /repeats));
            p.println("SD = " + SD);
            System.out.println("The time taken to select unique1 is "+(((double)sum) /repeats));
        }
            p.close();
            bw.close();
            fw.close();
           }
      catch(Exception ex) {
            System.out.println(ex);
            ex.printStackTrace();
      }
   }
}
```

```
// Searching for ten percent of objects without indexing

import org.neodatis.odb.core.Objects;
import org.neodatis.odb.main.ODB;
import org.neodatis.odb.main.ODBFactory;
import java.util.*;
import org.neodatis.odb.core.query.IQuery;
import org.neodatis.odb.core.query.criteria.CriteriaQuery;
import org.neodatis.odb.core.query.criteria.Where;
import java.io.*;
public class SearchNoIndexTenPercent {
   public static void main(String[] args) {
      ODB odb = null;
         try {
            System.out.println("Searching .....");
            FileWriter fw = new FileWriter("D:\\jim\\search-neodatis-no-index-tenPercent.txt");
            BufferedWriter bw = new BufferedWriter (fw);
            PrintWriter p = new PrintWriter(bw);
            int limit = 100000;
            for (int size = 5000; size <= limit; size+=5000) {
               long sum = 0;
               long sumSquares = 0;
               int repeats = 10;
               for(int m = 0; m < repeats; m++) {
                  long t1 = System.currentTimeMillis();
                  String ODB_NAME = ("neodatis-no-index-"+size+"-"+m);
                  odb = ODBFactory.open(ODB_NAME);
                  IQuery query = new CriteriaQuery(Item.class, Where.equal("tenPercent", 4));
                  Objects value = odb.getObjects(query);
                  long t2 = System.currentTimeMillis();
                  System.out.println("Items selected: "+value.size());
                  int i = 1;
                  while (value.hasNext()) {
                  System.out.println((i++)+ "\t:" + value.next()+ "\t");
                  }
                  odb.close();
               sum += (t2-t1);
               sumSquares+= (t2-t1)*(t2-t1);
               } //for int m
            System.out.println("Searched size = " + size + " For tenPercent = 4");
            double variance = ((double) (sumSquares - sum*sum/repeats) /repeats);
            double SD = Math.sqrt(variance);
            p.println("Searching time for size " + size + "\t" + (((double)sum) /repeats));
            p.println("SD = " + SD);
            System.out.println("The time taken to select unique1 is "+(((double)sum) /repeats));
         }
```

```
            p.close();
            bw.close();
            fw.close();
           }
       catch(Exception ex) {
            System.out.println(ex);
            ex.printStackTrace();
       }
   }
}

// Searching for twenty percent of objects without indexing

import org.neodatis.odb.core.Objects;
import org.neodatis.odb.main.ODB;
import org.neodatis.odb.main.ODBFactory;
import java.util.*;
import org.neodatis.odb.core.query.IQuery;
import org.neodatis.odb.core.query.criteria.CriteriaQuery;
import org.neodatis.odb.core.query.criteria.Where;
import java.io.*;

public class SearchNoIndexTwentyPercent {

   public static void main(String[] args) {
      ODB odb = null;
         try {
            System.out.println("Searching ........");
            FileWriter fw = new FileWriter("D:\\jim\\search-neodatis-no-index-
twentyPercent.txt");
            BufferedWriter bw = new BufferedWriter (fw);
            PrintWriter p = new PrintWriter(bw);
            int limit = 100000;
            for (int size = 5000; size <= limit; size+=5000) {
               long sum = 0;
               long sumSquares = 0;
               int repeats = 10;
               for(int m = 0; m < repeats; m++) {
                  long t1 = System.currentTimeMillis();
                  String ODB_NAME = ("neodatis-no-index-"+size+"-"+m);
                  odb = ODBFactory.open(ODB_NAME);
                  IQuery query = new CriteriaQuery(Item.class, Where.equal("twentyPercent",
2));
                  Objects value = odb.getObjects(query);
                  long t2 = System.currentTimeMillis();
                  System.out.println("Items selected: "+value.size());
                  int i = 1;
```

```
                while (value.hasNext()) {
                System.out.println((i++)+ "\t:" + value.next()+ "\t");
                }
                odb.close();
              sum += (t2-t1);
              sumSquares+= (t2-t1)*(t2-t1);
              } //for int m
            System.out.println("Searched size = " + size + " For twentyPercent = 2");
            double variance = ((double) (sumSquares - sum*sum/repeats) /repeats);
            double SD = Math.sqrt(variance);
            p.println("Searching time for size " + size + "\t" + (((double)sum) /repeats));
            p.println("SD = " + SD);
            System.out.println("The time taken to select unique1 is "+(((double)sum) /repeats));
        }
            p.close();
            bw.close();
            fw.close();
           }
      catch(Exception ex) {
            System.out.println(ex);
            ex.printStackTrace();
      }
   }
}


// Searching for fifty percent of objects without indexing

import org.neodatis.odb.core.Objects;
import org.neodatis.odb.main.ODB;
import org.neodatis.odb.main.ODBFactory;
import java.util.*;
import org.neodatis.odb.core.query.IQuery;
import org.neodatis.odb.core.query.criteria.CriteriaQuery;
import org.neodatis.odb.core.query.criteria.Where;
import java.io.*;

public class SearchNoIndexFiftyPercent {

   public static void main(String[] args) {
      ODB odb = null;
         try {
            System.out.println("Searching ........");
            FileWriter fw = new FileWriter("D:\\jim\\search-neodatis-no-index-
fiftyPercent.txt");
            BufferedWriter bw = new BufferedWriter (fw);
```

```
            PrintWriter p = new PrintWriter(bw);
            int limit = 100000;
            for (int size = 5000; size <= limit; size+=5000) {
               long sum = 0;
               long sumSquares = 0;
               int repeats = 10;
               for(int m = 0; m < repeats; m++) {
                  long t1 = System.currentTimeMillis();
                  String ODB_NAME = ("neodatis-no-index-"+size+"-"+m);
                  odb = ODBFactory.open(ODB_NAME);
                  IQuery query = new CriteriaQuery(Item.class, Where.equal("fiftyPercent", 1));
                  Objects value = odb.getObjects(query);
                  long t2 = System.currentTimeMillis();
                  System.out.println("Items selected: "+value.size());
                  int i = 1;
                  while (value.hasNext()) {
                  System.out.println((i++)+ "\t:" + value.next()+ "\t");
                  }
                  odb.close();
               sum += (t2-t1);
               sumSquares+= (t2-t1)*(t2-t1);
               } //for int m
            System.out.println("Searched size = " + size + " For fiftyPercent = 1");
            double variance = ((double) (sumSquares - sum*sum/repeats) /repeats);
            double SD = Math.sqrt(variance);
            p.println("Searching time for size " + size + "\t" + (((double)sum) /repeats));
            p.println("SD = " + SD);
            System.out.println("The time taken to select unique1 is "+(((double)sum) /repeats));
         }
            p.close();
            bw.close();
            fw.close();
           }
      catch(Exception ex) {
            System.out.println(ex);
            ex.printStackTrace();
      }
   }
}
```

A.3.3.1. Perst database

// Searching for the field *unique1*without indexing

```
import org.garret.perst.*;
import org.garret.perst.Storage;
import java.util.Iterator;
import java.util.*;
import org.garret.perst.StorageFactory;
import org.garret.perst.Storage;
import java.io.*;

public class SearchNoIndexUnique1 {
   public static void main (String[] args) {
        try {
           System.out.println("Searching .....");
           FileWriter fw = new FileWriter("D:\\jim\\search-perst-no-index-unique1.txt");
           BufferedWriter bw = new BufferedWriter (fw);
           PrintWriter p = new PrintWriter(bw);
           IPersistent[] list = null;
           Iterator iterator = null;
           int limit = 100000;
           for (int size = 5000; size <= limit; size+=5000) {
              long sum = 0;
              long sumSquares = 0;
              int repeats = 10;
              for(int m = 0; m < repeats; m++) {
                 long t1 = System.currentTimeMillis();
                 String dbname = "perst-no-index-" + size + "-" + m;
                 Storage db = StorageFactory.getInstance().createStorage();
                 db.open(dbname,Storage.INFINITE_PAGE_POOL);
                 IPersistentList root = (IPersistentList) db.getRoot();
                 Query query = db.createQuery();
                 query.prepare(Item.class, "unique1 = 300");
                 iterator = query.execute(root.iterator());
                 long t2 = System.currentTimeMillis();
                    while(iterator.hasNext()) {
                       System.out.println(iterator.next());
                    }
                 sum += (t2-t1);
                 sumSquares+= (t2-t1)*(t2-t1);
                 db.close();
              } // for int m
              System.out.println("Searched size = " + size + " For unique1 = 300");
              double variance = ((double) (sumSquares - sum*sum/repeats) /repeats);
```

```
                double SD = Math.sqrt(variance);
                p.println("Searching time for size " + size + "\t" + (((double)sum) /repeats));
                p.println("SD = " + SD);
                } // for int size
                p.close();
                bw.close();
                fw.close();
        }  // try
        catch(Exception ex) {
            System.out.println(ex);
            ex.printStackTrace();
        }
    }
}
```

// Searching for the field *unique2*without indexing

```
import org.garret.perst.*;
import org.garret.perst.Storage;
import java.util.Iterator;
import java.util.*;
import org.garret.perst.StorageFactory;
import org.garret.perst.Storage;
import java.io.*;

public class SearchNoIndexUnique2 {
    public static void main (String[] args) {
        try {
            System.out.println("Searching .....");
            FileWriter fw = new FileWriter("D:\\jim\\search-perst-no-index-unique2.txt");
            BufferedWriter bw = new BufferedWriter (fw);
            PrintWriter p = new PrintWriter(bw);
            IPersistent[] list = null;
            Iterator iterator = null;
            int limit = 100000;
            for (int size = 5000; size <= limit; size+=5000) {
                long sum = 0;
                long sumSquares = 0;
                int repeats = 10;
                for(int m = 0; m < repeats; m++) {
                    long t1 = System.currentTimeMillis();
                    String dbname = "perst-no-index-" + size + "-" + m;
                    Storage db = StorageFactory.getInstance().createStorage();
                    db.open(dbname,Storage.INFINITE_PAGE_POOL);
                    IPersistentList root = (IPersistentList) db.getRoot();
```

```
                Query query = db.createQuery();
                query.prepare(Item.class, "unique2 = 300");
                iterator = query.execute(root.iterator());
                long t2 = System.currentTimeMillis();
                   while(iterator.hasNext()) {
                      System.out.println(iterator.next());
                   }
                sum += (t2-t1);
                sumSquares+= (t2-t1)*(t2-t1);
                db.close();
             } // for int m
             System.out.println("Searched size = " + size + " For unique2 = 300");
             double variance = ((double) (sumSquares - sum*sum/repeats) /repeats);
             double SD = Math.sqrt(variance);
             p.println("Searching time for size " + size + "\t" + (((double)sum) /repeats));
             p.println("SD = " + SD);
             } // for int size
          p.close();
             bw.close();
             fw.close();
       }  // try
       catch(Exception ex) {
          System.out.println(ex);
          ex.printStackTrace();
       }
   }
}


// Searching for the field stringu1without indexing

import org.garret.perst.*;
import org.garret.perst.Storage;
import java.util.Iterator;
import java.util.*;
import org.garret.perst.StorageFactory;
import org.garret.perst.Storage;
import java.io.*;

public class SearchNoIndexStringu1 {
   public static void main (String[] args) {
      String x45 = "xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx";
        try {
           System.out.println("Searching .....");
           FileWriter fw = new FileWriter("D:\\jim\\search-perst-no-index-stringu1.txt");
           BufferedWriter bw = new BufferedWriter (fw);
```

```
            PrintWriter p = new PrintWriter(bw);
            IPersistent[] list = null;
            Iterator iterator = null;
            int limit = 100000;
            for (int size = 5000; size <= limit; size+=5000) {
               long sum = 0;
               long sumSquares = 0;
               int repeats = 10;
               for(int m = 0; m < repeats; m++) {
                  long t1 = System.currentTimeMillis();
                  String dbname = "perst-no-index-" + size + "-" + m;
                  Storage db = StorageFactory.getInstance().createStorage();
                  db.open(dbname,Storage.INFINITE_PAGE_POOL);
                  IPersistentList root = (IPersistentList) db.getRoot();
                  Query query = db.createQuery();
                  query.prepare(Item.class, "stringu1 = "'+convert(300)+ x45 +"'");
                  iterator = query.execute(root.iterator());
                  long t2 = System.currentTimeMillis();
                     while(iterator.hasNext()) {
                        System.out.println(iterator.next());
                              }
                  sum += (t2-t1);
                  sumSquares+= (t2-t1)*(t2-t1);
                  db.close();
               } // for int m
               System.out.println("Searched size = " + size + " For stringu1 = "'+convert(300)+
x45 +"'");
               double variance = ((double) (sumSquares - sum*sum/repeats) /repeats);
               double SD = Math.sqrt(variance);
               p.println("Searching time for size " + size + "\t" + (((double)sum) /repeats));
               p.println("SD = " + SD);
               } // for int size
               p.close();
               bw.close();
               fw.close();
         }  // try
         catch(Exception ex) {
            System.out.println(ex);
            ex.printStackTrace();
         }
   }

   private static String convert(int n) {
         /*****************************/
                  char[] result = new char[7];
                  char[] temp = new char[7];
```

```
                for(int k = 0; k < 7; k++) result[k] = 'z';
                int i = 6; int count = 0;
                while (n > 0) {
                        int rem = n % 26;
                        temp[i] = (char) ((int) 'A' + rem);
                        n = n / 26;
                        i--; count++;
                        }
                for(int k = 0; k < count; k++) result[k] = temp[++i];
                return(new String(result));
        }
}
```

// Searching for the field *stringu2*without indexing

```
import org.garret.perst.*;
import org.garret.perst.Storage;
import java.util.Iterator;
import java.util.*;
import org.garret.perst.StorageFactory;
import org.garret.perst.Storage;
import java.io.*;

public class SearchNoIndexStringu2 {
   public static void main (String[] args) {
      String x45 = "xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx";
         try {
            System.out.println("Searching .....");
            FileWriter fw = new FileWriter("D:\\jim\\search-perst-no-index-stringu2.txt");
            BufferedWriter bw = new BufferedWriter (fw);
            PrintWriter p = new PrintWriter(bw);
            IPersistent[] list = null;
            Iterator iterator = null;
            int limit = 100000;
            for (int size = 5000; size <= limit; size+=5000) {
               long sum = 0;
               long sumSquares = 0;
               int repeats = 10;
               for(int m = 0; m < repeats; m++) {
                  long t1 = System.currentTimeMillis();
                  String dbname = "perst-no-index-" + size + "-" + m;
                  Storage db = StorageFactory.getInstance().createStorage();
                  db.open(dbname,Storage.INFINITE_PAGE_POOL);
                  IPersistentList root = (IPersistentList) db.getRoot();
```

```
                Query query = db.createQuery();
                query.prepare(Item.class, "stringu2 = '" +convert(300)+ x45 + "'");
                iterator = query.execute(root.iterator());
                long t2 = System.currentTimeMillis();
                   while(iterator.hasNext()) {
                      System.out.println(iterator.next());
                                }
                sum += (t2-t1);
                sumSquares+= (t2-t1)*(t2-t1);
                db.close();
             } // for int m
             System.out.println("Searched size = " + size + " For stringu2 = '" +convert(300)+
x45 + "'");
             double variance = ((double) (sumSquares - sum*sum/repeats) /repeats);
             double SD = Math.sqrt(variance);
             p.println("Searching time for size " + size + "\t" + (((double)sum) /repeats));
             p.println("SD = " + SD);
             } // for int size
             p.close();
             bw.close();
             fw.close();
       }  // try
       catch(Exception ex) {
          System.out.println(ex);
          ex.printStackTrace();
       }
   }

   private static String convert(int n) {
       /*****************************/
             char[] result = new char[7];
             char[] temp = new char[7];
             for(int k = 0; k < 7; k++) result[k] = 'z';
             int i = 6; int count = 0;
             while (n > 0) {
                   int rem = n % 26;
                   temp[i] = (char) ((int) 'A' + rem);
                   n = n / 26;
                   i--; count++;
                   }
             for(int k = 0; k < count; k++) result[k] = temp[++i];
             return(new String(result));
       }
}
```

// Searching for the field *string4*without indexing

```
import org.garret.perst.*;
import org.garret.perst.Storage;
import java.util.Iterator;
import java.util.*;
import org.garret.perst.StorageFactory;
import org.garret.perst.Storage;
import java.io.*;

public class SearchNoIndexString4 {
   public static void main (String[] args) {
      String x45 = "xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx";
         try {
            System.out.println("Searching .....");
            FileWriter fw = new FileWriter("D:\\jim\\search-perst-no-index-string4.txt");
            BufferedWriter bw = new BufferedWriter (fw);
            PrintWriter p = new PrintWriter(bw);
            IPersistent[] list = null;
            Iterator iterator = null;
            int limit = 100000;
            for (int size = 5000; size <= limit; size+=5000) {
               long sum = 0;
               long sumSquares = 0;
               int repeats = 10;
               for(int m = 0; m < repeats; m++) {
                  long t1 = System.currentTimeMillis();
                  String dbname = "perst-no-index-" + size + "-" + m;
                  Storage db = StorageFactory.getInstance().createStorage();
                  db.open(dbname,Storage.INFINITE_PAGE_POOL);
                  IPersistentList root = (IPersistentList) db.getRoot();
                  Query query = db.createQuery();
                  query.prepare(Item.class, "string4 = 'AAAA"+x45+ "xxx'");
                  iterator = query.execute(root.iterator());
                  long t2 = System.currentTimeMillis();
                     while(iterator.hasNext()) {
                        System.out.println(iterator.next());
                              }
                  sum += (t2-t1);
                  sumSquares+= (t2-t1)*(t2-t1);
                  db.close();
               } // for int m
               System.out.println("Searched size = " + size + " For string4 = 'AAAA"+x45+
"xxx'");
               double variance = ((double) (sumSquares - sum*sum/repeats) /repeats);
               double SD = Math.sqrt(variance);
```

```
            p.println("Searching time for size " + size + "\t" + (((double)sum) /repeats));
            p.println("SD = " + SD);
            } // for int size
            p.close();
            bw.close();
            fw.close();
        }  // try
        catch(Exception ex) {
           System.out.println(ex);
           ex.printStackTrace();
        }
    }

    private static String convert(int n) {
        /*****************************/
                char[] result = new char[7];
                char[] temp = new char[7];
                for(int k = 0; k < 7; k++) result[k] = 'z';
                int i = 6; int count = 0;
                while (n > 0) {
                        int rem = n % 26;
                        temp[i] = (char) ((int) 'A' + rem);
                        n = n / 26;
                        i--; count++;
                        }
                for(int k = 0; k < count; k++) result[k] = temp[++i];
                return(new String(result));
        }
}


// Searching for one percent of objects without indexing

import org.garret.perst.*;
import org.garret.perst.Storage;
import java.util.Iterator;
import java.util.*;
import org.garret.perst.StorageFactory;
import org.garret.perst.Storage;
import java.io.*;

public class SearchNoIndexOnePercent {
    public static void main (String[] args) {
        try {
           System.out.println("Searching .....");
           FileWriter fw = new FileWriter("D:\\jim\\search-perst-no-index-onePercent.txt");
```

```
            BufferedWriter bw = new BufferedWriter (fw);
            PrintWriter p = new PrintWriter(bw);
            IPersistent[] list = null;
            Iterator iterator = null;
            int limit = 100000;
            for (int size = 5000; size <= limit; size+=5000) {
               long sum = 0;
               long sumSquares = 0;
               int repeats = 10;
               for(int m = 0; m < repeats; m++) {
                  long t1 = System.currentTimeMillis();
                  String dbname = "perst-no-index-" + size + "-" + m;
                  Storage db = StorageFactory.getInstance().createStorage();
                  db.open(dbname,Storage.INFINITE_PAGE_POOL);
                  IPersistentList root = (IPersistentList) db.getRoot();
                  Query query = db.createQuery();
                  query.prepare(Item.class, "onePercent = 1");
                  iterator = query.execute(root.iterator());
                  long t2 = System.currentTimeMillis();
                     while(iterator.hasNext()) {
                        System.out.println(iterator.next());
                              }
                  sum += (t2-t1);
                  sumSquares+= (t2-t1)*(t2-t1);
                  db.close();
               } // for int m
               System.out.println("Searched size = " + size + " For onePercent = 1");
               double variance = ((double) (sumSquares - sum*sum/repeats) /repeats);
               double SD = Math.sqrt(variance);
               p.println("Searching time for size " + size + "\t" + (((double)sum) /repeats));
               p.println("SD = " + SD);
               } // for int size
               p.close();
               bw.close();
               fw.close();
         }  // try
         catch(Exception ex) {
            System.out.println(ex);
            ex.printStackTrace();
         }
      }
   }
```

```
// Searching for ten percent of objects without indexing

import org.garret.perst.*;
import org.garret.perst.Storage;
import java.util.Iterator;
import java.util.*;
import org.garret.perst.StorageFactory;
import org.garret.perst.Storage;
import java.io.*;
public class SearchNoIndexTenPercent {
   public static void main (String[] args) {
        try {
           System.out.println("Searching .....");
           FileWriter fw = new FileWriter("D:\\jim\\search-perst-no-index-tenPercent.txt");
           BufferedWriter bw = new BufferedWriter (fw);
           PrintWriter p = new PrintWriter(bw);
           IPersistent[] list = null;
           Iterator iterator = null;
           int limit = 100000;
           for (int size = 5000; size <= limit; size+=5000) {
              long sum = 0;
              long sumSquares = 0;
              int repeats = 10;
              for(int m = 0; m < repeats; m++) {
                 long t1 = System.currentTimeMillis();
                 String dbname = "perst-no-index-" + size + "-" + m;
                 Storage db = StorageFactory.getInstance().createStorage();
                 db.open(dbname,Storage.INFINITE_PAGE_POOL);
                 IPersistentList root = (IPersistentList) db.getRoot();
                 Query query = db.createQuery();
                 query.prepare(Item.class, "tenPercent = 7");
                 iterator = query.execute(root.iterator());
                 long t2 = System.currentTimeMillis();
                    while(iterator.hasNext()) {
                       System.out.println(iterator.next());
                                }
                 sum += (t2-t1);
                 sumSquares+= (t2-t1)*(t2-t1);
                 db.close();
              } // for int m
              System.out.println("Searched size = " + size + " For tenPercent = 7");
              double variance = ((double) (sumSquares - sum*sum/repeats) /repeats);
              double SD = Math.sqrt(variance);
              p.println("Searching time for size " + size + "\t" + (((double)sum) /repeats));
              p.println("SD = " + SD);
              } // for int size
```

```
            p.close();
            bw.close();
            fw.close();
        }  // try
        catch(Exception ex) {
            System.out.println(ex);
            ex.printStackTrace();
        }
    }}


// Searching for twenty percent of objects without indexing

import org.garret.perst.*;
import org.garret.perst.Storage;
import java.util.Iterator;
import java.util.*;
import org.garret.perst.StorageFactory;
import org.garret.perst.Storage;
import java.io.*;

public class SearchNoIndexTwentyPercent {
    public static void main (String[] args) {
        try {
            System.out.println("Searching .....");
            FileWriter fw = new FileWriter("D:\\jim\\search-perst-no-index-twentyPercent.txt");
            BufferedWriter bw = new BufferedWriter (fw);
            PrintWriter p = new PrintWriter(bw);
            IPersistent[] list = null;
            Iterator iterator = null;
            int limit = 100000;
            for (int size = 5000; size <= limit; size+=5000) {
                long sum = 0;
                long sumSquares = 0;
                int repeats = 10;
                for(int m = 0; m < repeats; m++) {
                    long t1 = System.currentTimeMillis();
                    String dbname = "perst-no-index-" + size + "-" + m;
                    Storage db = StorageFactory.getInstance().createStorage();
                    db.open(dbname,Storage.INFINITE_PAGE_POOL);
                    IPersistentList root = (IPersistentList) db.getRoot();
                    Query query = db.createQuery();
                    query.prepare(Item.class, "twentyPercent = 3");
                    iterator = query.execute(root.iterator());
                    long t2 = System.currentTimeMillis();
                        while(iterator.hasNext()) {
```

```
                        System.out.println(iterator.next());
                                }
                    sum += (t2-t1);
                    sumSquares+= (t2-t1)*(t2-t1);
                    db.close();
                } // for int m
                System.out.println("Searched size = " + size + " For twentyPercent = 3");
                double variance = ((double) (sumSquares - sum*sum/repeats) /repeats);
                double SD = Math.sqrt(variance);
                p.println("Searching time for size " + size + "\t" + (((double)sum) /repeats));
                p.println("SD = " + SD);
                } // for int size
                p.close();
                bw.close();
                fw.close();
            }  // try
            catch(Exception ex) {
                System.out.println(ex);
                ex.printStackTrace();
            }
    }
}


// Searching for fifty percent of objects without indexing

import org.garret.perst.*;
import org.garret.perst.Storage;
import java.util.Iterator;
import java.util.*;
import org.garret.perst.StorageFactory;
import org.garret.perst.Storage;
import java.io.*;

public class SearchNoIndexFiftyPercent {
    public static void main (String[] args) {
            try {
                System.out.println("Searching .....");
                FileWriter fw = new FileWriter("D:\\jim\\search-perst-no-index-fiftyPercent.txt");
                BufferedWriter bw = new BufferedWriter (fw);
                PrintWriter p = new PrintWriter(bw);
                IPersistent[] list = null;
                Iterator iterator = null;
                int limit = 100000;
                for (int size = 5000; size <= limit; size+=5000) {
                    long sum = 0;
```

```
            long sumSquares = 0;
            int repeats = 10;
            for(int m = 0; m < repeats; m++) {
               long t1 = System.currentTimeMillis();
               String dbname = "perst-no-index-" + size + "-" + m;
               Storage db = StorageFactory.getInstance().createStorage();
               db.open(dbname,Storage.INFINITE_PAGE_POOL);
               IPersistentList root = (IPersistentList) db.getRoot();
               Query query = db.createQuery();
               query.prepare(Item.class, "fiftyPercent = 1");
               iterator = query.execute(root.iterator());
               long t2 = System.currentTimeMillis();
                  while(iterator.hasNext()) {
                     System.out.println(iterator.next());
                              }
               sum += (t2-t1);
               sumSquares+= (t2-t1)*(t2-t1);
               db.close();
            } // for int m
            System.out.println("Searched size = " + size + " For fiftyPercent = 1");
            double variance = ((double) (sumSquares - sum*sum/repeats) /repeats);
            double SD = Math.sqrt(variance);
            p.println("Searching time for size " + size + "\t" + (((double)sum) /repeats));
            p.println("SD = " + SD);
            } // for int size
            p.close();
            bw.close();
            fw.close();
         }  // try
         catch(Exception ex) {
            System.out.println(ex);
            ex.printStackTrace();
         }
   }
}
```

A.4. Java code for modifying objects in the databases

A.4.1. Db4o database

A.4.1.1. Modifying objects without indexing enabled

```
import java.util.*;
import com.db4o.Db4o;
```

```
import com.db4o.ObjectContainer;
import com.db4o.ObjectSet;
import com.db4o.query.*;
import java.io.*;

public class ModifyNoIndex {

   public static void main(String[] args) {
      ObjectContainer db = null;
      try {
            System.out.println("Modifying ......");
            FileWriter fw = new FileWriter("D:\\jim\\modify-db4o-no-index.txt");
            BufferedWriter bw = new BufferedWriter (fw);
            PrintWriter p = new PrintWriter(bw);
            int limit = 100000;
            for (int size = 5000; size <= limit; size+=5000) {
               long sum = 0;
               long sumSquares = 0;
               int repeats = 10;
               long t1 = System.currentTimeMillis();
               for (int m = 0; m<repeats; m++) {
                  String dbname = ("db4o-no-index-"+size+"-"+m);
                  db = Db4o.openFile(dbname);
                  Db4o.configure().objectClass(Item.class).cascadeOnUpdate(true);
                  Query query = db.query();
                  query.constrain(Item.class);
                  query.descend("unique1").constrain(new Integer(300));
                  ObjectSet result = query.execute();
                     for(int k=0; k<result.size(); k++) {
                        System.out.println(result);
                        Item found = (Item)result.next();
                        found.addNumber(k);
                        db.set(found);
                        db.commit();
                        System.out.println("Added "+k+ " to "+found);
                     }
                  db.close();
               }
               long t2 = System.currentTimeMillis();
               sum += (t2-t1);
               sumSquares+= (t2-t1)*(t2-t1);
               System.out.println("Modified size = " + size + "For unique1 = 300");
               double variance = ((double) (sumSquares - sum*sum/repeats) /repeats);
               double SD = Math.sqrt(variance);
               p.println("Modifying time for size " + size + "\t" + (((double)sum) /repeats));
               p.println("SD = " + SD);
```

```
            } // for int size
            p.close();
            bw.close();
            fw.close();
         }
         catch(Exception ex) {
            System.out.println(ex);
            ex.printStackTrace();
      }

   }}
```

A.4.2. Neodatis database

A.4.2.1. Modifying objects without indexing enabled

```
import org.neodatis.odb.core.Objects;
import org.neodatis.odb.main.ODB;
import org.neodatis.odb.main.ODBFactory;
import java.util.*;
import org.neodatis.odb.core.query.IQuery;
import org.neodatis.odb.core.query.criteria.CriteriaQuery;
import org.neodatis.odb.core.query.criteria.Where;
import java.io.*;


public class ModifyNoIndex {
   public static String ODB_NAME = null;

   public static void main(String[] args) {
      ODB odb = null;
         try {
            System.out.println("Modifying.....");
            FileWriter fw = new FileWriter("D:\\jim\\modify-neodatis-no-index.txt");
            BufferedWriter bw = new BufferedWriter (fw);
            PrintWriter p = new PrintWriter(bw);
            int limit = 100000;
            for (int size = 5000; size <= limit; size+=5000) {
               long sum = 0;
               long sumSquares = 0;
               long t1 = System.currentTimeMillis();
               int repeats = 10;
               for (int m = 0; m<repeats; m++) {
                  ODB_NAME = ("neodatis-no-index-"+size+"-"+m);
                  odb = ODBFactory.open(ODB_NAME);
```

```
                IQuery query = new CriteriaQuery(Item.class, Where.equal("unique1",300));
                Objects value = odb.getObjects(query);
                System.out.println("Items selected "+value.size());
                for (int k = 0; k<value.size(); k++) {
                   Item found = (Item)value.next();
                   found.addNumber(k);
                   odb.store(found);
                   odb.commit();
                   System.out.println("Added "+k+ " to "+found);
                }
                odb.close();
             } //for int m
             long t2 = System.currentTimeMillis();
             sum += (t2-t1);
             sumSquares+= (t2-t1)*(t2-t1);
             System.out.println("Modified size = " + size + "For unique1 = 300");
             double variance = ((double) (sumSquares - sum*sum/repeats) /repeats);
             double SD = Math.sqrt(variance);
             p.println("Modifying time for size " + size + "\t" + (((double)sum) /repeats));
             p.println("SD = " + SD);
          } //for int size
          p.close();
          bw.close();
          fw.close();
          } //for try

      catch(Exception ex) {
            System.out.println(ex);
      }
   }
}
```

A.4.3. Perst database

A.4.3.1. Modifying objects without indexing enabled

```
import org.garret.perst.*;
import org.garret.perst.Storage;
import java.util.Iterator;
import java.util.*;
import org.garret.perst.StorageFactory;
import org.garret.perst.Storage;
import java.io.*;

public class ModifyNoIndex {
```

```
public static void main (String[] args) {
      Item[] items = new Item[100000];
      try {
         System.out.println("Modifying ......");
         FileWriter fw = new FileWriter("D:\\jim\\modify-perst-no-index.txt");
         BufferedWriter bw = new BufferedWriter (fw);
         PrintWriter p = new PrintWriter(bw);
         IPersistent[] list = null;
         int limit = 100000;
         for (int size = 5000; size <= limit; size+=5000) {
            long sum = 0;
            long sumSquares = 0;
            long t1 = System.currentTimeMillis();
            int repeats = 10;
            for(int m = 0; m < repeats; m++) {
               String dbname = "perst-no-index-" + size + "-" + m;
               Storage db = StorageFactory.getInstance().createStorage();
               db.open(dbname,Storage.INFINITE_PAGE_POOL);
               IPersistentList root = (IPersistentList) db.getRoot();
                     Query query = db.createQuery();
                        query.prepare(Item.class, "unique1 = 300");
                        Iterator iter = query.execute(root.iterator());
               while (iter.hasNext()) {
                        System.out.println(iter.next());
              }
               iter = query.execute(root.iterator());
               int index = 0;
               while (iter.hasNext())items[index++] = (Item) iter.next();
               for(int n = 0; n < index; n++) {
                  root.remove(items[n]);
                  items[n].num = 1;
                  items[n].modify();
                  root.add(items[n]);
               }
               iter = query.execute(root.iterator());
               while (iter.hasNext()) {
                        System.out.println(iter.next());
               }
               db.close();
            }  // for int m
            long t2 = System.currentTimeMillis();
            sum += (t2-t1);
            sumSquares+= (t2-t1)*(t2-t1);
            System.out.println("Modified size = " + size + "For unique1 = 300");
            double variance = ((double) (sumSquares - sum*sum/repeats) /repeats);
            double SD = Math.sqrt(variance);
```

```
            p.println("Modifying time for size " + size + "\t" + (((double)sum) /repeats));
            p.println("SD = " + SD);
            } // for int size
            p.close();
            bw.close();
            fw.close();
        }  // try
        catch(Exception ex) {
            System.out.println(ex);
            ex.printStackTrace();
        }
    }}
```

A.5. Java code for deleting objects in the databases

A.5.1. Db4o database

A.5.1.1. Deleting objects without indexing enabled

```
import java.util.*;
import com.db4o.Db4o;
import com.db4o.ObjectContainer;
import com.db4o.ObjectSet;
import com.db4o.query.*;
import java.io.*;

public class DeleteNoIndex {
    public static void main(String[] args) {

    ObjectContainer db = null;

    try {
        System.out.println("Deleting.....");
        FileWriter fw = new FileWriter("D:\\jim\\delete-db4o-noindex.txt");
        BufferedWriter bw = new BufferedWriter (fw);
        PrintWriter p = new PrintWriter(bw);
        int limit = 100000;
        for (int size = 5000; size <= limit; size+=5000) {
            long sum = 0;
            long sumSquares = 0;
            String dbname = ("db4o-no-index"+size+"-"+0);
            db = Db4o.openFile(dbname);
            Db4o.configure().objectClass(Item.class).cascadeOnDelete(true);
            long t1 = System.currentTimeMillis();
            int repeats = 10;
```

```
            for(int m = 0; m<repeats; m++) {
               Query query = db.query();
               query.constrain(Item.class);
               query.descend("unique1").constrain(new Integer(300));
               ObjectSet result = query.execute();
               for(int k=0; k<result.size(); k++) {
                  Item found = (Item)result.next();
                  db.delete(found);
                  db.commit();
                  System.out.println("Deleted: "+found);
               }
            long t2 = System.currentTimeMillis();
            sum += (t2-t1);
            sumSquares+= (t2-t1)*(t2-t1);

            System.out.println("Deleted size = " + size + "For unique1 = 300");
            double variance = ((double) (sumSquares - sum*sum/repeats) /repeats);
            double SD = Math.sqrt(variance);
            p.println("Deleting time for size " + size + "\t" + (((double)sum) /repeats));
            p.println("SD = " + SD);
            } // for int size
            p.close();
            bw.close();
            fw.close();
         }
      }
            catch(Exception ex) {
               System.out.println(ex);
               ex.printStackTrace();
            }
   }
}
```

A.5.2. Neodatis database

A.5.2.1. Deleting objects without indexing enabled

```
import org.neodatis.odb.core.Objects;
import org.neodatis.odb.main.ODB;
import org.neodatis.odb.main.ODBFactory;
import java.util.*;
import org.neodatis.odb.core.query.IQuery;
import org.neodatis.odb.core.query.criteria.CriteriaQuery;
import org.neodatis.odb.core.query.criteria.Where;
```

```
import java.io.*;

public class DeleteNoIndex {
   public static String ODB_NAME = null;

   public static void main(String[] args) {
      ODB odb = null;

         try {
            System.out.println("Deleting ........");
            FileWriter fw = new FileWriter("D:\\jim2\\delete-neodatis-no-index.txt");
            BufferedWriter bw = new BufferedWriter (fw);
            PrintWriter p = new PrintWriter(bw);
            int limit = 100000;
            for (int size = 5000; size <= limit; size+=5000) {
            ODB_NAME = ("neodatis-no-index-"+size+"-"+limit);
            odb = ODBFactory.open (ODB_NAME);
            IQuery query = new CriteriaQuery(Item.class, Where.equal("onePercent",5));
            Objects value = odb.getObjects(query);
            System.out.println("Items selected "+value.size());
            long t7 = System.currentTimeMillis();
            for (int k = 0;k<value.size();k++) {
               Item found = (Item)value.next();
               odb.delete(found);
               odb.commit();
               System.out.println("Deleted: "+found);
            }
            long t8 = System.currentTimeMillis();
            long Time4 = t8-t7;
            System.out.println("The time taken to delete one percent is "+Time4);
         }
            p.close();
            bw.close();
            fw.close();
            if(odb!=null) {
               odb.close();
            }
         }
      catch(Exception ex) {
            System.out.println(ex);
      }
   }
}
```

A.5.3. Perst database

A.5.3.1. Deleting objects without indexing enabled

```
import org.garret.perst.*;
import org.garret.perst.Storage;
import java.util.Iterator;
import java.util.*;

import org.garret.perst.StorageFactory;
import org.garret.perst.Storage;
import java.io.*;

public class DeleteNoIndex {
   public static void main (String[] args) {

         try {
            System.out.println("Deleting ......");
            FileWriter fw = new FileWriter("D:\\jim\\delete-perst-noindex.txt");
            BufferedWriter bw = new BufferedWriter (fw);
            PrintWriter p = new PrintWriter(bw);
            IPersistent[] list = null;
            int limit = 100000;
            for (int size = 5000; size <= limit; size+=5000) {
               long sum = 0;
               long sumSquares = 0;
               long t1 = System.currentTimeMillis();
               int repeats = 10;
               for(int m = 0; m < repeats; m++) {
                  String dbname = "perst-noindex-" + size + "-" + m;
                  Storage db = StorageFactory.getInstance().createStorage();
                  db.open(dbname,Storage.INFINITE_PAGE_POOL);
                  IPersistentList root = (IPersistentList) db.getRoot();
                                    Query query = db.createQuery();
                                    query.prepare(Item.class, "unique1 = 300");
                                    Iterator iter = query.execute(root.iterator());
                  while (iter.hasNext()) {
                           System.out.println(iter.next());
                        }
                  while (iter.hasNext()) {
                     Item item = (Item) iter.next();
                     root.remove(item);
                     item.num = 1;
                     item.modify();
                     root.add(item);
```

```
                    }   // for n
                    iter = query.execute(root.iterator());
                    while (iter.hasNext()) {
                            System.out.println(iter.next());
                    }
                    db.close();
                }  // for int m
                long t2 = System.currentTimeMillis();
                sum += (t2-t1);
                sumSquares+= (t2-t1)*(t2-t1);
                System.out.println("Deleted size = " + size + "For unique1 = 300");
                double variance = ((double) (sumSquares - sum*sum/repeats) /repeats);
                double SD = Math.sqrt(variance);
                p.println("Deleting time for size " + size + "\t" + (((double)sum) /repeats));
                p.println("SD = " + SD);
                } // for int size
                p.close();
                bw.close();
                fw.close();
        }  // try
        catch(Exception ex) {
            System.out.println(ex);
            ex.printStackTrace();
        }
    }
}
```

A.5.4. Prevayler database

A.5.4.1 Deleting objects without indexing enabled

```
import org.prevayler.Prevayler;
import org.prevayler.PrevaylerFactory;
import java.util.*;
import java.io.*;

public class DeleteNoIndexUnique1 {
    public static void main(String[] args) {

        Prevayler prevayler = null;
        ItemKeeper itemKeeper = null;
        try {
            System.out.println("Deleting .....");
            FileWriter fw = new FileWriter("D:\\jim\\delete-prevayler-no-index-unique1-a.txt");
            BufferedWriter bw = new BufferedWriter (fw);
```

```
        PrintWriter p = new PrintWriter(bw);
        int limit = 100000;
        for(int size = 5000; size<=limit; size+=5000) {
           long sum = 0;
           long sumSquares = 0;
           int repeats = 10;
              for(int m = 0; m < repeats; m++) {
                 long t1 = System.currentTimeMillis();
                 String dbname = ("prevayler-no-index-a-"+size+"-"+m);
                 prevayler = PrevaylerFactory.createPrevayler(new ItemKeeper(), dbname);
                 itemKeeper = (ItemKeeper)prevayler.prevalentSystem();
                 List list = null;
                    for(int k = 0; k<99; k++) {
                       list = (List) prevayler.execute(new SelectUnique1(300));
                    }
                 for(int k = 0;k<list.size();k++) {
                    Item found = (Item)list.get(k);
                    list.remove(found);
                 }
                 long t2 = System.currentTimeMillis();
                 sum += (t2-t1);
                 sumSquares+= (t2-t1)*(t2-t1);
              }
           System.out.println("Deleted size = " + size + " For unique1 = 300");
           double variance = ((double) (sumSquares - sum*sum/repeats) /repeats);
           double SD = Math.sqrt(variance);
           p.println("Deleting time for size " + size + "\t" + (((double)sum) /repeats));
           p.println("SD = " + SD);
           prevayler.close();
        }
        p.close();
        bw.close();
        fw.close();
     }
     catch(Exception ex) {
           System.out.println(ex);
           ex.printStackTrace();
     }


  }
}
```

Appendix **B**

B.1. Times for creating databases and standard deviations

	10000	20000	30000	40000	50000	60000	70000	80000	90000	100000
Db4o (Time)	636	1406	2403	3463	4542	4803	5670	6981	8043	12621
Db4o (SD)	44.8	72.4	329.3	686.5	866	267.3	319.8	797.6	1659.2	3537
Neodatis (Time)	657	1415	2471	3684	5456	5644	10529	9354	9621	9681
Neodatis (SD)	73.6	152.2	124.1	200.4	927.7	435.6	3260.6	1540.4	1848.4	679.2
Perst (Time)	1.5	7.8	9.3	12.3	17.2	26.7	26.6	40.6	59.2	43.9
Perst (SD)	4.5	7.8	7.6	6.1	4.9	10	6.9	10.6	13.8	11.8
Prevayler (Time)	111228	221454	335532	450326	556387	667170	797795	902915	1012612	1119829
Prevayler (SD)	2923.7	5583.4	7228.2	6387.8	12000.7	14776.6	42662.8	12357.3	8236.2	25181.1

B.2. Times for searching for *unique2* without indexing

	10000	20000	30000	40000	50000	60000	70000	80000	90000	100000
Db4o (Time)	353.1	2315.7	2293.8	2387.5	4681.2	5028.1	3871.9	4821.9	6365.7	6970.3
Db4o (SD)	74.2	217.4	435.1	329.3	1335.7	2058.6	299.4	605.3	1956.4	1854.3
Neodatis (Time)	882.8	1612.5	2729.7	4901.6	14321.8	11803.1	10714.2	12935.9	16750	23559.4
Neodatis (SD)	238.6	331.8	717.6	692.4	15562	9853.5	3721.2	4136.1	9193	28156
Perst (Time)	6.2	4.7	4.7	3.1	3.1	6.3	4.7	3.1	1.6	3.1
Perst (SD)	7.6	7.1	7.1	6.2	6.2	7.7	7.1	6.2	4.8	6.2
Prevayler (Time)	1320.3	3232.8	4187.5	5784.4	6937.5	8715.6	12662.5	11250	14706.2	16771.8
Prevayler (SD)	114.3	654.4	117.1	280.1	152.1	879.8	2526.2	733.5	2780.6	4459.1

B.3. Times for searching for one percent of objects without indexing

	10000	20000	30000	40000	50000	60000	70000	80000	90000	100000
Db4o (Time)	2920.2	4486.1	2984.1	7407.7	5814	3899.9	9059.5	4940.7	10036	6959.4
Db4o (SD)	2490.2	3846.8	1392.3	7687	4758.9	407.3	9992.2	757.6	8299.1	1820
Neodatis (Time)	887.6	1837.5	3107.9	6229.6	11767.2	8706.3	14779.7	15311.1	10495.5	13576.4
Neodatis (SD)	243.4	419.1	1326.2	2405.4	17114.4	3157.4	15603.7	4372	1578.1	3201.9
Perst (Time)	6.3	4.7	4.7	4.6	4.7	3.2	3.1	4.7	3.1	3.1
Perst (SD)	7.7	7.1	7.1	7	7.1	6.4	6.2	7.1	6.2	6.2
Prevayler (Time)	1317.2	3292.2	4181.3	5854.7	7595.3	8725	12528.2	11706.2	14014.1	16050
Prevayler (SD)	69.1	492.5	91.1	312.5	599.3	528.7	1060	554.1	941.6	608.6

B.4. Times for searching for ten percent of objects without indexing

	10000	20000	30000	40000	50000	60000	70000	80000	90000	100000
Db4o (Time)	914.2	4556.3	2414.1	2768.6	3211	4170.3	4018.8	6486	6817.3	8232.6
Db4o (SD)	62.4	2770.7	1017.8	857.5	304.5	583.5	724.4	1632.9	3288	2270.4
Neodatis (Time)	899.9	1650	2531.2	4553	5529.6	6448.3	9373.4	11665.8	10268.4	10715.5
Neodatis (SD)	239.8	190	302.4	666.9	436.7	1395.8	2291.7	1994.7	1534.4	1681.6
Perst (Time)	6.3	4.7	4.7	3.1	3.2	4.7	4.7	3.1	3.1	3.1
Perst (SD)	7.7	7.1	7.1	6.2	6.4	7.1	7.1	6.2	6.2	6.2
Prevayler (Time)	1325	2870.3	5042.2	5776.6	7025	8421.9	10103.1	11221.9	12759.3	14306.3
Prevayler (SD)	74.6	115.5	1132.2	188.2	234.4	378.2	796.9	191.1	947.6	872.2

B.5. Times for searching for twenty percent of objects without indexing

	10000	20000	30000	40000	50000	60000	70000	80000	90000	100000
Db4o (Time)	923.4	2623.4	2087.5	2188.9	3568.6	5470.1	3990.5	4758	5779.8	6565.9
Db4o (SD)	157.5	555.4	449.4	280.3	1104.7	4332.8	590.1	354.1	805.7	1592.3
Neodatis (Time)	956.2	1923.3	2498.5	5004.6	6054.6	6382.9	9228.1	11114	11578.2	11650
Neodatis (SD)	239.5	475.8	265.5	1186.3	1248.3	975.5	1181.2	1669.1	1957.5	1306.5
Perst (Time)	4.7	3.1	4.7	3.1	3.1	3.1	4.7	3.1	4.6	3.1
Perst (SD)	7.1	6.2	7.1	6.2	6.2	6.25	7.1	6.2	7	6.2
Prevayler (Time)	1514.1	3753.1	4985.9	8471.9	8650	10712.5	13929.7	11678.1	13862.5	15932.8
Prevayler (SD)	68.7	321.4	309.4	2081.7	626	1151.8	2835	391.1	1062	1704.4

B.6. Times for searching for fifty percent of objects without indexing

	10000	20000	30000	40000	50000	60000	70000	80000	90000	100000
Db4o (Time)	293.8	2198.6	1985.8	2181.2	4337.4	3600.2	3897	5004.8	5420.2	7257.9
Db4o (SD)	69	170	455.5	192.3	2361.8	356	458.9	832	669.7	3788.1
Neodatis (Time)	956.2	1923.3	2498.5	5004.6	6054.6	6382.9	9228.1	11114	11578.2	11650
Neodatis (SD)	239.5	475.8	265.5	1186.3	1248.3	975.5	1181.2	1669.1	1957.5	1306.5
Perst (Time)	6.3	4.6	4.7	4.7	4.6	3.1	4.7	3.2	3.1	3.2
Perst (SD)	7.7	7	7.1	7.1	7	6.2	7.1	6.4	6.2	6.4
Prevayler (Time)	1751.6	3117.2	4695.3	6689.1	8137.5	13635.9	13065.6	15665.7	17367.2	16567.1
Prevayler (SD)	242	156.3	173.1	801.6	1287.6	2961.4	1987.7	1998.9	3986.6	2650.4

B.7. Times for searching for field *unique2* less than 1000

	10000	20000	30000	40000	50000	60000	70000	80000	90000	100000
Db4o (Time)	873.5	2089.2	1892.2	2161.1	4096.9	7687.5	3695.2	5465.6	8975.2	6204.7
Db4o (SD)	80.6	62.7	133.7	197.1	2015.8	7612.97	171.9	2986.7	8702.7	1578.1
Neodatis (Time)	940.6	1740.8	2364	7743.6	5827.9	5759.4	9865.6	12132.7	10050.1	11975
Neodatis (SD)	289.6	655.6	243.5	7334	938.2	957.7	4968.6	3021.3	1706.5	4169.6
Perst (Time)	67.1	84.7	57.8	62.5	75	59.5	61	56.5	82.9	54.6
Perst (SD)	17.3	14.2	9.9	9.85	6	9.6	14.7	15.8	17.3	12.6
Prevayler (Time)	1468.8	3689.1	5501.6	5781.2	7064	8292.2	10703.1	11109.3	12859.4	14332.8
Prevayler (SD)	139.2	389	807. 6	338.3	440.1	247.3	1595.8	595.3	1262.5	625.1

B.8. Times for searching for field *unique1* less than 1000

	10000	20000	30000	40000	50000	60000	70000	80000	90000	100000
Db4o (Time)	898.2	2172.2	2212.4	2296.9	3276.6	3893.6	3886.1	8051.5	5389.1	6710.8
Db4o (SD)	75.7	97.9	720.8	214.9	318.5	662.3	298	5433	368	1812.1
Neodatis (Time)	898.5	1629.8	2642.1	4509.4	5920.3	5893.7	8081.5	10678	10334.4	9771.8
Neodatis (SD)	219.2	220	1141.7	535.3	1372	903.6	535.2	2805.2	1902.7	1020.2
Perst (Time)	34.3	102.9	70.4	71.7	79.9	75	79.5	59.5	76.6	53.1
Perst (SD)	30.1	28.9	18.9	19.9	19	34.7	14.6	11.9	14.8	14.4
Prevayler (Time)	1257.8	2710.9	4273.4	6457.8	7289.1	10176.5	9712.5	11976.6	12798.4	14445.4
Prevayler (SD)	73.7	81.6	233.5	1635.1	335.8	2435.3	359.6	2445.7	1406.6	1465

B.9. Times for searching for field *stringu1* without indexing

	10000	20000	30000	40000	50000	60000	70000	80000	90000	100000
Db4o (Time)	906.3	2196.8	2086	2326.3	3226.6	3956.3	3742.4	4656.3	5501.7	7059.5
Db4o (SD)	134.9	144.2	490.4	363.3	377.7	551.3	164.3	434	735.4	4055.7
Neodatis (Time)	173.4	1665.6	2543.8	4781.2	6632.8	6231.2	11134.3	10110.9	14046.9	11331.2
Neodatis (SD)	11.1	270.5	275.5	634.6	4002.6	1051.9	6401.3	962.4	5988.9	1842.1
Perst (Time)	6.3	4.7	4.7	4.7	3.1	3.1	3.1	3.1	3.1	4.7
Perst (SD)	7.7	7.1	7.1	7.1	6.2	6.2	6.2	6.2	6.2	7.1
Prevayler (Time)	1434.4	3056.2	4765.6	6062.5	7837.5	9620.3	10929.7	12687.5	13935.9	15471.8
Prevayler (SD)	63.6	104.8	482	54.1	909.8	358.3	1003.7	866.2	521.5	776.4

B.10. Times for searching for field *stringu2* without indexing

	10000	20000	30000	40000	50000	60000	70000	80000	90000	100000
Db4o (Time)	881.3	2135.8	1949.9	2084.5	3149.9	5262.7	3731.4	4849.9	7723.5	7123.5
Db4o (SD)	79.9	85.9	152.7	96.6	327.5	2206.3	129.6	597.1	6815.1	1863.7
Neodatis (Time)	176.6	1621.8	2486	4989.1	5804.6	6101.6	8060.9	10611	10032.8	12673.5
Neodatis (SD)	12.1	188.2	221.9	640.9	675.9	1315.6	862.6	1589.8	1532	6455.2
Perst (Time)	6.3	3.1	4.7	4.7	3.1	6.2	4.7	3.1	3.1	4.7
Perst (SD)	7.7	6.25	7.1	7.1	6.2	7.6	7.1	6.2	6.2	7.1
Prevayler (Time)	1426.6	3070.3	4782.8	6325	9471.8	9190.7	11326.6	16000	15311	17828.1
Prevayler (SD)	67.5	35.1	199.1	227.8	3260.7	245.4	916.7	5386.5	1147.3	1194.9

B.11. Times for searching for field *string4* without indexing

	10000	20000	30000	40000	50000	60000	70000	80000	90000	100000
Db4o (Time)	887.3	2112.3	1818.6	2064.2	3623.4	3772.1	3780	4825	5843.9	6768.5
Db4o (SD)	72.6	89.9	66.9	117.8	621.2	574.3	160.4	377.5	909.5	1784.2
Neodatis (Time)	429.8	2092.3	2946.9	5346.9	6514.1	6559.4	9085.9	12014	12154.7	12935.9
Neodatis (SD)	364.1	358.4	347.7	677	1043.7	884.5	1015.1	2389.3	1664.3	1287.8
Perst (Time)	6.2	4.7	4.7	3.2	4.7	3.1	4.6	3.1	3.1	3.2
Perst (SD)	7.6	7.1	7.1	6.4	7.1	6.2	7	6.2	6.2	6.4
Prevayler (Time)	1392.2	2826.6	4201.5	5503.1	12153.1	10196.9	16442.2	11071.9	12301.6	13718.8
Prevayler (SD)	75.2	169.8	128.2	95.3	3166.3	2813.3	3853	892	517.1	711.6

B.12. Times for searching for field *unique1* with indexing

	10000	20000	30000	40000	50000	60000	70000	80000	90000	100000
Db4o (Time)	223.3	92.2	82.8	57.6	67.3	155.9	118.7	148.5	93.8	146.8
Db4o (SD)	77.4	19.1	72	15.5	11.9	45.2	30.5	58.3	70.2	108
Neodatis (Time)	1596.9	2848.5	5259.3	5742.2	7634.3	12768.7	12893.8	18059.4	17651.6	17945.3
Neodatis (SD)	447.2	622.9	719.2	1156.1	1070.5	6542.4	4252.3	10249.9	3083.8	3741.84
Perst (Time)	47	26.6	24.7	28.2	18.8	18.8	26.6	30.2	23.9	14.2
Perst (SD)	22	13.9	14.4	11.5	11.5	11.5	13.9	10.8	12.7	10.8
Prevayler (Time)	-	-	-	-	-	-	-	-	-	-
Prevayler (SD)	-	-	-	-	-	-	-	-	-	-

B.13. Times for searching for field *stringu1* with indexing

	10000	20000	30000	40000	50000	60000	70000	80000	90000	100000
Db4o (Time)	59.4	56.2	103.3	50.1	76.5	57.9	61.1	104.7	76.6	57.8
Db4o (SD)	9.6	17.3	52.5	6.2	68.9	7.1	4.7	85.4	69.9	12.2
Neodatis (Time)	1603.1	2725	10395.3	5492.2	7954.7	10671.9	13131.3	16187.5	16042.3	18239.1
Neodatis (SD)	450.1	605.7	5819.3	993.9	1005.1	1070.1	2599.1	3829.4	2038.4	2550.4
Perst (Time)	37.4	21.9	23.8	32.9	41.1	30.1	25.1	23.5	27.8	31
Perst (SD)	20.9	14.2	16.1	14.7	17.3	17.7	14.2	12.4	11.9	12.3
Prevayler (Time)	-	-	-	-	-	-	-	-	-	-
Prevayler (SD)	-	-	-	-	-	-	-	-	-	-

B.14. Times for modifying a unique record with indexing

	10000	20000	30000	40000	50000	60000	70000	80000	90000	100000
Db4o (Time)	520.3	596.8	629.7	682.8	529.6	621.8	568.8	587.5	586	359.4
Db4o (SD)	1560.9	1790.4	1889.1	2048.4	1588.8	1865.4	1706.4	1762.5	1758	1078.2
Neodatis (Time)	1501.6	2715.6	5767.2	5909.4	7531.3	12979.7	16051.5	21701.6	18228.2	25037.5
Neodatis (SD)	4504.8	8146.8	17301.6	17728.2	22593.9	38939.1	48154.5	65104.8	54684.6	75112.5
Perst (Time)	321.9	220.3	232.8	340.6	246.8	232.8	312.5	214	251.5	228.1
Perst (SD)	965.7	660.9	698.4	1021.8	740.4	698.4	937.5	642	754.5	684.3
Prevayler (Time)	-	-	-	-	-	-	-	-	-	-
Prevayler (SD)	-	-	-	-	-	-	-	-	-	-

B.15. Times for deleting a single item without indexing

	10000	20000	30000	40000	50000	60000	70000	80000	90000	100000
Db4o (Time)	0	1.6	0	1.5	0	0	1.6	0	1.6	0
Db4o (SD)	0	4.8	0	4.5	0	0	4.8	0	4.8	0
Neodatis (Time)	1009.3	1590.6	2614.1	4906.3	8329.7	8567.2	13934.4	10104.7	10739.1	10100
Neodatis (SD)	3027.9	4771.8	7842.3	14718.9	24989.1	25701.6	41803.2	30314.1	32217.3	30300
Perst (Time)	1070.3	2468.7	1945.3	3814.1	3095.3	4228.2	6070.3	4625	9237.5	7845.3
Perst (SD)	3210.9	7406.1	5835.9	11442.3	9285.9	12684.6	18210.9	13875	27712.5	23535.9
Prevayler (Time)	1251.6	2818.7	4271.9	5415.6	6737.5	10760.9	9693.7	11578.2	12386	13878.1
Prevayler (SD)	86.3	115.5	233	37.1	119.2	2725.6	405.6	934.3	608.9	1267.5

B.16. Times for deleting 1000 randomly distributed items without indexing

	10000	20000	30000	40000	50000	60000	70000	80000	90000	100000
Db4o (Time)	0	0	1.5	0	0	0	0	1.6	0	1.6
Db4o (SD)	0	0	4.5	0	0	0	0	4.8	0	4.8
Neodatis (Time)	3037.5	3960.9	6489	9037.5	10811	10653.1	15818.8	28928.1	21196.8	21504.7
Neodatis (SD)	9112.5	11882.7	19467	27112.5	32433	31959.3	47456.4	86784.3	63590.4	64514.1
Perst (Time)	1090.6	2065.6	1893.7	2650	4948.4	3487.5	4178.1	3992.2	6637.5	8732.9
Perst (SD)	3271.8	6196.8	5681.1	7950	14845.2	10462.5	12534.3	11976.6	19912.5	26198.7
Prevayler (Time)	1339.1	2806.2	4525	5506.3	6773.4	8762.5	10814.1	10812.5	12826.6	13703.1
Prevayler (SD)	31.3	98	718.1	145.5	71	851.8	2420	140.2	1586.2	553.7

B.17. Times for deleting 1000 randomly distributed items with indexing

	10000	20000	30000	40000	50000	60000	70000	80000	90000	100000
Db4o (Time)	0	0	1.5	0	0	0	0	1.6	0	1.6
Db4o (SD)	0	0	4.5	0	0	0	0	4.8	0	4.8
Neodatis (Time)	10.9	12.5	7.8	7.8	9.4	4.7	4.7	7.8	6.2	7.8
Neodatis (SD)	32.7	37.5	23.4	23.4	28.2	14.1	14.1	23.4	18.6	23.4
Perst (Time)	4286	6811	8573.5	8045.3	9470.3	8942.2	9970.3	7856.3	12115.6	7353.1
Perst (SD)	12858	20433	25720.5	24135.9	28410.9	26826.6	29910.9	23568.9	36346.8	22059.3
Prevayler (Time)	-	-	-	-	-	-	-	-	-	-
Prevayler (SD)	-	-	-	-	-	-	-	-	-	-