Balancing simplicity and efficiency in Web applications

FINAL REPORT

By Marc Hull (mfh02)
Supervised by Peter McBrien

http://www.doc.ic.ac.uk/~mfh02/project
14th June 2006
ABSTRACT

Static websites are a thing of the past. These days, even the simplest of sites tend to include some dynamic content, whether it is a message board, photo gallery or online shopping service. However, writing dynamic websites is often a trade-off between simplicity and efficiency. Scripting languages provide a shallow learning curve and fast development cycle, but are often too slow for use in enterprise environments. Meanwhile, enterprise tools allow for highly efficient websites to be written, but have complex interfaces and place restrictions on the program design.

This report describes the development of a tool that provides both simplicity and efficiency in web applications. The persistence of state between web pages is highlighted as the main source of problems with the majority of existing tools and is improved through the provision of a system for persisting objects regardless of their type, termed an “orthogonal persistence” mechanism, for the Java language. In this system, any Java object can be persisted to a relational database, with all the required SQL being generated and executed automatically. Several techniques for improving efficiency using this technique are implemented, which optimise the generated SQL based on the usage patterns of persistent objects. Further improvements are achieved through the use of a presentation system that provides a clear separation between page content and style and a server that automatically detects and applies changes made to dynamic web pages, recreating the instant update features of scripting languages.
Acknowledgements

I would firstly like to thank my supervisor, Dr Peter McBrien, for his support and enthusiasm for this project throughout the year. His insight into database techniques generated many interesting discussions during the course of this project which helped to shape many of the optimisation techniques used.

I would also like to acknowledge the following people for various contributions:

- Professor Susan Eisenbach for her support of this project and help in finding a willing supervisor for it.
- Dr Paul Kelly for suggesting suitable areas of research and tools covered in this project.
- Dr Tony Field for several discussions in which potential directions for this project were highlighted.
- Hani Suleiman, Anthony Mak and many other people at Formicary for filling in my project questionnaire and providing an enterprise perspective.
- Tristan Allwood and Francis Russell for interesting discussions about the Java language.
Balancing simplicity and efficiency in Web applications

1 INTRODUCTION .............................................................................................................. 9
1.1 OBJECTIVES .............................................................................................................. 10
1.2 CONTRIBUTIONS ...................................................................................................... 11
1.3 REPORT OUTLINE ................................................................................................. 11

2 BACKGROUND ............................................................................................................. 13
2.1 PERSISTENT STORES .............................................................................................. 14
  2.1.1 Serialization ......................................................................................................... 14
  2.1.2 Relational Databases .......................................................................................... 15
  2.1.3 Object-Relational Databases .......................................................................... 16
  2.1.4 Object-Oriented Databases ............................................................................ 16
  2.1.5 Object-Relational Mapping ............................................................................. 17
  2.1.6 Performance Evaluation .................................................................................... 22
2.2 PERSISTENCE LAYERS ........................................................................................... 29
  2.2.1 Java Database Connectivity (JDBC), ActiveX Data Objects (ADO)............... 30
  2.2.2 iBatis SQL Maps ............................................................................................... 30
  2.2.3 Enterprise Java Beans 2 ................................................................................... 31
  2.2.4 Java Data Objects (JDO) .................................................................................. 33
  2.2.5 Hibernate ......................................................................................................... 36
  2.2.6 Enterprise Java Beans 3 .................................................................................. 38
2.3 RELATED RESEARCH .............................................................................................. 39
  2.3.1 PJava .................................................................................................................. 39
  2.3.2 CDuce ............................................................................................................... 40
  2.3.3 C# 3.0 .............................................................................................................. 41
  2.3.4 Haskell List Comprehensions ......................................................................... 41
  2.3.5 Nestor .............................................................................................................. 42
2.4 PRESENTATION LAYERS ......................................................................................... 43
  2.4.1 HTML and CSS .................................................................................................. 43
  2.4.2 Java Server Pages (JSPs) .................................................................................. 43
  2.4.3 WebWork .......................................................................................................... 44
  2.4.4 SiteMesh ........................................................................................................... 45
  2.4.5 XML and XSLT .................................................................................................. 45
  2.4.6 Tapestry ............................................................................................................ 45
2.5 CONCLUSIONS ......................................................................................................... 46

3 REQUIREMENTS ELICITATION ..................................................................................... 51
3.1 CASE STUDY ............................................................................................................. 52
  3.1.1 About The Web Application .............................................................................. 52
  3.1.2 Test Strategy ..................................................................................................... 52
  3.1.3 Announcements Module .................................................................................. 53
  3.1.4 Calendar Module .............................................................................................. 54
3.2 QUESTIONNAIRE ..................................................................................................... 56
  3.2.1 Table And Query Optimisation ....................................................................... 56
  3.2.2 Separation Of Content And Formatting ........................................................... 57
  3.2.3 Prioritising Of Requirements ........................................................................... 59
  3.2.4 Target Language ............................................................................................... 60
3.3 PROTOTYPE ................................................................................................................ 61
  3.3.1 Semantics Of Using The Collection API ........................................................... 62
  3.3.2 Object Updates .................................................................................................. 63
  3.3.3 Database Garbage Collection .......................................................................... 64
3.4 JAVA AND THE JVM .................................................................................................. 65
  3.4.1 Reference Semantics ....................................................................................... 65
  3.4.2 Field Modifiers .................................................................................................. 67
  3.4.3 Garbage Collection ........................................................................................... 67
  3.4.4 Inheritance ....................................................................................................... 70
4 SPECIFICATION..........................................................................................73
4.1 THE PERSISTENCE LAYER ........................................................................................................74
  4.1.1 Overview.................................................................................................................................74
  4.1.2 Persistence Interface ..................................................................................................................75
  4.1.3 Observational Equivalence.......................................................................................................75
  4.1.4 Persistable Types.....................................................................................................................75
  4.1.5 Maintaining Java Semantics.....................................................................................................79
  4.1.6 The Heuristic Optimiser ..........................................................................................................81
  4.1.7 Maintenance Task....................................................................................................................82
4.2 THE PRESENTATION LAYER .........................................................................................................82
  4.2.1 Web Page Interface ..................................................................................................................83
  4.2.2 Persistent Data Interface .........................................................................................................83
  4.2.3 Document Interface ................................................................................................................84
4.3 THE SERVER LAYER ...................................................................................................................86
  4.3.1 Loading of Web Applications .................................................................................................86
  4.3.2 Interception HTTP Requests .................................................................................................87
  4.3.3 User Interface ........................................................................................................................87
5 THE PERSISTENCE LAYER .........................................................................................................89
  5.1 OVERVIEW ...............................................................................................................................90
  5.2 OBJECT STORAGE .....................................................................................................................91
    5.2.1 Storage Of Class Instances .....................................................................................................91
    5.2.2 Storage Of Primitives And Autoboxed Primitives ................................................................101
    5.2.3 Storage Of Arrays And Lists ..................................................................................................103
    5.2.4 Storage Of Sets And Maps ....................................................................................................110
  5.3 OBJECT RETRIEVAL ..................................................................................................................110
    5.3.1 Retrieval Of Class Instances .................................................................................................110
    5.3.2 Retrieval Of Primitives .........................................................................................................121
    5.3.3 Retrieval Of Arrays And Lists .............................................................................................122
    5.3.4 Retrieval Of Sets And Maps ..................................................................................................123
  5.4 OPTIMISATIONS .........................................................................................................................123
    5.4.1 Resolving Generic Types .......................................................................................................123
    5.4.2 Optimisation Heuristics .......................................................................................................126
    5.4.3 Schema Change ....................................................................................................................128
  5.5 SUMMARY ..................................................................................................................................129
6 THE PRESENTATION LAYER .........................................................................................................131
  6.1 ANNOTATING PAGES .................................................................................................................131
  6.2 HANDLING WEB DATA .............................................................................................................132
  6.3 DOCUMENT TREE ......................................................................................................................133
  6.4 CONCLUSIONS ...........................................................................................................................135
7 THE SERVER LAYER ....................................................................................................................137
  7.1 REQUESTDELEGATINGSERVICE .............................................................................................137
  7.2 WEBAPPLICATIONMANAGER .................................................................................................138
  7.3 CONFIGURATIONMANAGER ....................................................................................................139
  7.4 CONCLUSIONS ...........................................................................................................................140
8 EVALUATION ..............................................................................................................................141
  8.1 PERSISTENCE LAYER ...............................................................................................................141
    8.1.1 Provision Of Orthogonal Persistence ......................................................................................141
    8.1.2 Overhead Of Automatic Persistence ....................................................................................145
    8.1.3 Lazy/Eager Loading Comparison .........................................................................................147
8.1.4 Schema Optimisation Heuristics................................................................. 150
8.2 THE PRESENTATION LAYER........................................................................... 152
8.3 THE SERVER LAYER....................................................................................... 152
8.4 SUMMARY....................................................................................................... 153

9 CONCLUSIONS ................................................................................................. 155
9.1 CONTRIBUTIONS OF THIS PROJECT........................................................... 156
9.2 FUTURE WORK................................................................................................. 157
  9.2.1 Prefetch Paths.............................................................................................. 157
  9.2.2 Heuristic Cost Function Improvements..................................................... 158
  9.2.3 HTML Import.............................................................................................. 158
9.3 SUMMARY....................................................................................................... 159

10 BIBLIOGRAPHY............................................................................................... 161

11 APPENDIX........................................................................................................ 163
  11.1 MORM² REPRESENTATION........................................................................... 163
  11.2 PERSISTENT STORE TESTS.......................................................................... 165
    11.2.1 Test Environment..................................................................................... 165
    11.2.2 Results Gathering..................................................................................... 165
    11.2.3 Insert Test With Small Objects................................................................. 165
    11.2.4 Select Test For A Single Small Object..................................................... 168
    11.2.5 Select Test For Multiple Small Objects.................................................... 169
    11.2.6 Memory Usage Test With Small Objects................................................ 171
    11.2.7 Insert Test With Large Objects................................................................. 172
    11.2.8 Select Test For A Single Large Object (1)................................................. 174
    11.2.9 Select Test For A Single Large Object (2)................................................ 176
    11.2.10 Select Test For Multiple Large Objects................................................ 177
    11.2.11 Insert Test For Class Hierarchies............................................................ 179
    11.2.12 Select Test For Multiple Objects From A Class Hierarchy (1)............... 181
    11.2.13 Select Test For Multiple Objects From A Class Hierarchy (2)............... 182
    11.2.14 Memory Usage Test With Class Hierarchies.......................................... 184
  11.3 CASE STUDY TESTS...................................................................................... 185
    11.3.1 Announcements Module............................................................................ 185
    11.3.2 Calendar Module....................................................................................... 189
  11.4 REQUIREMENTS QUESTIONNAIRE............................................................ 194
1 Introduction

Dynamic web content has become increasingly essential for even the smallest of websites in recent years. Personal sites now often contain blogs, message boards and photo galleries that can be updated by many different people all over the world. Business sites make use of complex content management systems to improve their relationships with their clients and share resources between offices that are thousands of miles apart. Also, the recent spread of web services has allowed many companies and individuals to bring together previously disparate technologies in new and interesting ways.

However, while the uses of web pages have advanced significantly in the last 10 years, the same improvements do not seem to have been made to the back-end tools that make them work. Many different frameworks for writing dynamic web applications now exist, but most still require low-level tasks to be performed such as specifying an object-relational mapping for any persistent data and writing separate HTML templates for each page. With the functionality provided by web applications becoming increasingly complex, such time-consuming activities now form a large part of a web developer’s working day. This project addresses this by first settling on a high-level interface for web development interface that promotes reuse of components and imposes minimal restrictions on the design of the web application, and then finding ways of optimising this system to improve the overall performance of web applications using it.

These goals are achieved by breaking up the role of a web application framework into three separate layers:

- The **persistence layer** is responsible for maintaining persistent state between web requests, typically by storing data in a relational database.
- The **presentation layer** provides methods for generating web page content and style for display in a client web browser.
- The **server layer** intercepts HTTP requests received from the client and is responsible for loading and invoking the appropriate components for servicing those requests.

For each of these layers, a number of existing tools were examined to find their strengths and weaknesses, then areas of improvement were highlighted and a number of design and implementation strategies tested. For the persistence layer, a system for automatically generating and executing the queries required to persist arbitrary Java objects was provided, which allows web application developers to alter the data storage requirements of their applications with minimal impact to their application code. A number of query optimisation techniques were also applied to improve the overall efficiency of storing and accessing objects. The presentation layer was designed to allow code for constructing individual web pages to be reused by completely separating page content from design. Finally, the server layer was constructed to detect changes to a web application and dynamically load the altered classes.

**Figure 1-1** shows a high-level view of the overall architecture of this project containing the tasks and interactions for each of the layers.
1.1 Objectives

The high-level objectives of this project are as follows:

- To provide a system of orthogonal persistence for the Java language in which:
  - Objects can be persisted regardless of their type.
  - The semantics of the Java language are maintained for persistent objects.
  - No modifications to the Java Virtual Machine are required.
  - Efficient access to persisted objects is provided.
- To provide a presentation layer in which:
  - Pages of content can be constructed independently of the format in which they are displayed to the user.
  - Different types of web data can be handled by a consistent interface.
- To provide a server layer in which:
  - HTTP requests are intercepted and delegated to the relevant web application.
  - Changes to the class or source files of a web application are detected and automatically compiled and loaded into the running system.
  - A user interface allows web applications to be added, removed and configured.
1.2 Contributions

The main contributions of the project are as follows:

- A functional persistence system is provided that supports the efficient storage and retrieval of instances of a large number of Java data types while preserving the semantics of the Java language. This is achieved without modifying the Java Virtual Machine.

- Several techniques for improving access to persistent objects are implemented and evaluated. Lazy loading of field data is achieved without prior knowledge of the classes being persisted. Heuristics for optimising the object-relational mapping are used to improve overall performance for particular patterns of persistent object usage.

- An extensible system for representing and transforming web page content is provided which allows the same content to be viewed in multiple different formats. Developers can easily create their own page components using a simple content tree and style transformation interface.

- A system for simulating the instant update functionality of scripting languages is provided for Java. Changes to web application code are monitored and altered source files are automatically compiled and loaded as required.

1.3 Report Outline

The remainder of this report is organised as follows:

- The background section highlights existing tools and technologies that can be used in the development of a web application. The persistence and presentation layers of various web application frameworks are compared, and a review of relevant research is given to describe the current state-of-the-art in this area.

- The requirements elicitation section describes a number of ways in which the specification of this project was formed. The findings of a case study, questionnaire, prototype and assessment of the Java language are discussed.

- The specification section outlines a set of high-level goals that are then refined to low-level requirements for each layer of the project.

- Sections on each layer describe the evolution of design and implementation throughout the course of the project. Various problems that were not initially apparent in the specification are discussed and solutions proposed.

- The evaluation section assesses the success of the project by comparing the implementation of each layer to the high-level and low-level goals set out in the specification section.

- The conclusions section summarises the achievements of the project, its contributions towards current research in the area and proposes topics for future work.
2 Background

A web application is a piece of software that is accessible via a network connection and can service multiple concurrent users. In the last ten years, web applications have become popular due to their ability to provide worldwide access to some centralised resource via the Internet from any computer that can render HTML web pages. This has lead to them being widely used for e-commerce, online banking, file sharing and other lucrative businesses. More recent developments in web services and a drive to shift more and more client functionality into web applications show that they are still a very active area in computing.

Often such applications are divided into three layers that each provides a different piece of functionality:

The presentation layer accepts user requests, gathers the content necessary for the response, formats it in some appropriate way and then returns it to the user.

The business layer defines the objects used for storing data in the application and controls the operations that can be performed upon this data.

The persistence layer handles the storage and retrieval of business objects so that data is not lost between executions of the application.

Typically, a web application framework will provide services to minimise the effort developers have to put in to the presentation and persistence layers of their application. This project attempts to analyse and improve the role of the framework within these areas, and so is not concerned with the business layer which is often customised to the specific domain of each application.

Much investment and research has already gone in to the development of efficient, maintainable web applications. This section starts by summarising and evaluating a number of different strategies used to implement persistence. Existing products and research projects that have attempted to provide persistence services are then covered to give an indication of past work and the current state-of-the-art. Presentation layer systems are then discussed, followed by conclusions drawn from all of the products discussed and potential improvements to them that could be implemented as part of this project.
2.1 Persistent Stores

Keller [Keller 04] defines persistence as “the ability of an object to survive the lifecycle of the process in which it resides”, which could be used to refer to a number of existing strategies. Typically, useful persistence systems save objects to some form of non-volatile storage so that they can survive machine crashes. They also follow the CRUD pattern; that is, objects may be created, read, updated and deleted from their store as necessary.

This section describes and compares the three common methods of object persistence: serialization, relational databases and object-oriented databases. It also covers two extensions to the relational model used to provide persistence of objects: object-relational databases and object-relational mappings.

2.1.1 Serialization

Modern languages such as Java and those in .NET support serialization as part of the core platform ([Sun 01], [MSDN 05]) so that developers can easily store and retrieve objects without the need to write custom serialization algorithms. User-defined classes can be marked as serializable and instances can then be passed to various library classes that allow them to be written out to files or sent over network sockets.

The main advantage of this approach is its ease of use, since it allows almost any class to be converted into a storable form and then recovered at a later date. However, it is often unsuitable for large web applications due to its inability to scale efficiently for large object graphs. This is caused by the algorithm described in Figure 1, “Default serialization behaviour of Java for reference fields”, which allows only complete object graphs to be persisted and retrieved, causing the big inhale problem.

Serialization suffers badly in comparison to the access times and memory usage of relational or object-oriented databases, which allow complex queries to be specified that retrieve or update only the required data. However, using it in combination with a relational database to provide near-orthogonal

---

1 In Java this is done by altering the class to implement the empty Serializable interface, while .NET requires classes to carry the Serializable annotation.
orthogonal persistence\textsuperscript{2} with improved querying, update and indexing is a possibility and is used in the basic object-relational mapping technique described in [Ananthanarayanan et al. 93].

1. The object to be serialized is chosen
2. Non-static and non-transitive field references from that object are followed to form the reachability graph
3. The reachable object graph is saved to a stream (e.g. a file)
4. On deserialization, the entire reachable object graph is reconstructed

\textbf{Figure 2-2} Default serialization behaviour of Java for reference fields

\subsection{2.1.2 Relational Databases}

\textit{Relational databases} provide an efficient mechanism for storing and retrieving large volumes of structured data. Originally proposed in 1970 [Codd 70], they have since benefited from years of investment and research, which have caused a large increase in their efficiency and feature set. Prior to the proliferation of object-oriented languages, relational databases were seen as the de facto standard for storing data and are therefore used widely in industry.

However, relational databases offer a view of data as unordered sets (or bags) of tuples in relations, where each tuple is composed of primitive values such as text or numbers. Objects, on the other hand, may contain references to other objects and be part of a class hierarchy that determines the data they inherit from other classes. This difference in data models, called the \textit{impedance mismatch} ([Carey and DeWitt 96]), does not make relational databases ideally suited to storing objects. Often developers have to add a lot of scaffolding code to their classes in order to convert their object data into a form accepted by the relational table schema and back again.

Several attempts have been made to reduce the impedance mismatch between objects and relational databases. The SQL 1999 standard allows for user-defined, nested types with accessor and mutator methods [Eisenberg and Melton 99]. It also addresses the notion of object identity, by giving rows a unique identifier that can be referenced using the special REF data type.

\textsuperscript{2} The term \textit{orthogonal persistence} is used as described in [Atkinson et al. 96] to mean persistence for all data irrespective of their type. However, in this report it is taken to only include the persistence of object data and not object code.
CREATE TYPE emp_type UNDER person_type
AS ( emp_id INTEGER, salary REAL )
INSTANTIABLE NOT FINAL
REF ( emp_id )
INSTANCE METHOD give_raise
( abs_or_pct BOOLEAN, amount REAL )
RETURNS REAL

Unfortunately, many DBMS vendors have chosen not to fully implement the SQL 1999 standard, with the features mentioned above missing from major RDBMSs such as PostgreSQL, Microsoft SQL Server and MySQL. An improved version of the standard, SQL 2003 [Eisenberg et al. 04], was released to address some of the issues raised by vendors concerning the 1999 version, but it also has yet to gain wide acceptance.

2.1.3 Object-Relational Databases

Before the SQL 1999 standard was released, many database vendors attempted to add their own object support to relational databases, producing what are known as object-relational databases. Typically these provide a few features on top of an existing relational database to simplify queries over more abstract data structures. Since these features emerged through vendors rather than from a standard, the actual features provided vary greatly between implementations, but often user-defined types and functions are allowed. They also tend to allow tables to be related as part of inheritance hierarchies or through references and provide implicit joins between these tables, as demonstrated by navigating the department reference in Figure 2-4.

SELECT firstName, lastName FROM student, department WHERE department.id = student.department AND department.name = "Computing"

Figure 2-3 Example of how table reference traversal is achieved in relational databases

SELECT firstName, lastName FROM student WHERE student.department.name = "Computing"

Figure 2-4 Example of table reference traversal in an object-relational database

Many relational database systems offer some object-relational features, including PostgreSQL, Oracle and Microsoft SQL Server. However, most of these features are typically also provided by object-relational mapping layers (discussed later in section 2.1.5), which reduce the need for user-defined types and automatically generated JOINs. As such, object-relational features tend to be used as a shorthand for those accessing databases directly, and are not considered further in this report.

2.1.4 Object-Oriented Databases

Object-oriented databases (OODBs) were introduced in an attempt to eliminate the impedance mismatch problem experienced when persisting objects to relational databases, while still providing the performance and memory benefits of using a database for persistence. User-defined types containing references are supported as first-class citizens in the database, allowing objects to be moved seamlessly between transient and persisted states.

As well as providing a more elegant interface for users to persist their objects, OODBs were also expected to surpass the performance of relational databases when accessing them. Join operations, a common performance hit when traversing object references in relational databases, could be greatly reduced in OODBs by using physical addresses as object identifiers [Kim 93].
However, despite heavy research into object-oriented databases from the mid 80s through to the mid 90s, they failed to take off in the way many had anticipated [Carey and DeWitt 96]. Unlike relational databases, which had benefited greatly from a well-specified data model and the quick emergence of SQL as the standard query language, the definition of an Object-Oriented Database was less clear. Many vendors had already began writing their own commercial products before the components that make up an OODB were even agreed upon in 1992 [Atkinson et al. 92], and it wasn't until 1994 that the Object Database Management Group (ODMG) was formed and put together the first standards for OODBs [Cattell 94].

Object oriented databases also suffer from a more fundamental problem that makes them less attractive to many businesses. In order to completely eliminate the impedance mismatch suffered when persisting objects to relational databases, OODBs are often locked in to a particular language. Additionally, their promised performance gains compared to relational databases are often overshadowed by the lack of features, including good querying facilities and transaction support.

![Worldwide database revenue by category for 2002 based on new license sales (millions of dollars)](image)

**Figure 2-5** Worldwide Database Revenue by Category for 2002 Based on New License Sales (Millions of Dollars). Source: Gartner Dataquest (June 2003)

### 2.1.5 Object-Relational Mapping

Drawing upon the continued success of relational databases, object-relational mappings provide a layer between the relational database and object-oriented programming language which aims to eliminate the impedance mismatch between the two. Although the mapping will be tied to the language it is designed for, the actual persisted data are still independent of it, allowing them to be accessed via custom mappings for other languages. Additionally, existing legacy data in relational databases can still be used with this technique, making it attractive to businesses that have already invested in relational technology.
In order to successfully persist objects from an object-oriented language to a relational database, an O-R mapper must construct table schema that will support the efficient storage and retrieval of object graphs and inheritance hierarchies. A number of different solutions exist for both of these problems.

In the following mapping strategies, the graphical representation used is based on that of the M²ORM²+HIE model. For more information about this representation, please see section 11.1 of the Appendix.

### 2.1.5.1 Object Graph Persistence

Object-oriented languages allow object members to be references to other objects, forming an object graph. For a persisted object to be fully reconstructed when retrieved from a persistent store, all referenced objects must also have been persisted. The simplest solution to this is described by Ananthanarayanan as Basic Mapping [Ananthanarayanan et al. 93]. W Keller also mentions two other common approaches to this problem; Single Table Aggregation and Foreign Key Aggregation [W Keller 97].

For each different object graph mapping, the following example will be used:

### Basic Mapping

In this solution, each object is stored in serialized form as a single value in a relational tuple. To retrieve the object, that value would be read from the database and be de-serialized by the language from which it came. The database knows nothing of the structure of the object's data, so queries over the object's fields cannot be performed, partial objects cannot be fetched and the data are tied to the language in which they were serialized. However, this can be the most memory efficient method for storing objects and may offer performance benefits in some cases.
corresponding Engine and Chassis object for each Car instance is efficient, since they can be retrieved with the Car object without an additional lookup.

**Single Table Aggregation**

As its name suggests, this approach uses one table and flattens an object tree into a single schema containing all the members of all the objects involved. Like basic mapping, this allows an entire section of the object graph to be reconstructed from a single row of the table, but also allows the fields to be included in queries.

![Single table aggregation mapping example](image)

However, this solution cannot cater for cyclic references since only object trees can be successfully flattened. It also cannot distinguish between null references and objects whose fields may all be null without needing an additional flag column. Care must also be taken when modifying objects that are referenced from multiple places, since the object's values will be duplicated in the table. A unique object ID column can be stored with each reference to allow all references to the same object to be updated simultaneously.

**Foreign Key Aggregation**

![Foreign key aggregation mapping example](image)
This method closely follows the logical structure of objects by using foreign keys as references between rows. Each class is mapped to a separate table and each instance of a class is uniquely identified by an artificial object ID integer (oid). References are mapped to integer columns containing either the oid of the object they reference or NULL. This overcomes the problems associated with single table aggregation, but can suffer from poor performance. Navigating references of an object now requires either multiple SELECT statements or a JOIN.

2.1.5.2 Inheritance Hierarchy Persistence

As well as being related to other classes through references, classes may be related through their inheritance hierarchy. The main concern to the persistence system is to allow queries that are executed over a class to encompass instances of any subclasses and to ensure that the runtime types of persisted objects are preserved. As with object graph persistence, there are a number of strategies for achieving this, each offering different performance or memory gains based on how the persisting application uses its data.

Many sources describe the three common forms of inheritance mapping under differing names and with minor variations [A Keller et al. 93] [W Keller 97] [Hibernate 05]. These strategies are described below as Table Per Class, Table Per Inheritance Path and Table Per Inheritance Hierarchy. It is also worth noting that the basic mapping scheme described above can also be used to persist objects that belong to class hierarchies, but it does not allow for querying over the hierarchy.

For each different inheritance hierarchy mapping, the following example will be used:

![Figure 2-10 UML representation of inheritance hierarchy example](image)

**Table Per Class**

In this strategy, each class in the hierarchy is mapped to a separate table schema which stores the member variables defined in that class plus an oid field. When an object is persisted, a unique oid value is generated, then a row is inserted into each table associated with the runtime class and all superclasses of the object, with each row sharing the generated oid value. To retrieve an object, all rows sharing that object's oid value must be fetched, either through multiple SELECT statements or a JOIN.

For performance reasons, a discriminator column may be added to each table which stores the name of the runtime class of each row. This is to cater for the case when a single object is being fetched using a query over one of its superclass tables. The discriminator column will reveal the true runtime type of the object, allowing only the tables along that branch of the inheritance tree to be included in the JOIN operation needed to reconstruct the complete object.
Table Per Inheritance Path

In this solution, a separate table schema is used for each class as in the table per class approach, but these tables now contain fields for the member variables of all superclasses. When an object is persisted, only a single row needs to be added into the table associated with that object's runtime class. To fetch a particular object when the runtime class is known, only a single `SELECT` need be executed on the table associated with that class. However, queries over superclasses would require all subclass tables to be accessed. In this method, abstract classes do not need to map to separate tables since they cannot form concrete objects. Queries over an abstract class simply need to be performed across the tables of all concrete subclasses of that class. This is described as the table per concrete class approach in Hibernate [Hibernate 05].
Table Per Inheritance Hierarchy

A single table can be used to store an entire inheritance hierarchy by flattening the tree of member variables across all classes of the hierarchy. All objects are then persisted as single rows in this table, with NULL values for fields of member variables which are not in their runtime class. A discriminator column must be added to keep track of the runtime classes of objects and to distinguish between member variables that do not exist and that are simply null references.

This approach can offer good performance since only a single database operation is needed to add or retrieve an object, however it can also waste a large amount of space with many NULL values for deep hierarchies. Persisting very many objects to a single table can also negatively affect performance due to locking, since many users might attempt to access and update the contents of the table concurrently. Because of these considerations, this strategy is often only useful for small sections of the inheritance hierarchy where instances of each class are often retrieved together.

![Diagram of Table Per Inheritance Hierarchy](image)

2.1.6 Performance Evaluation

As well as ease of development, the performance of a persistent store can greatly affect its suitability in different situations. To find out which type of persistent store would be most suitable for a web application, I ran a series of tests on a number of existing products and compared their results. For each test, the following products were used:

- Java Serialization
- PostgreSQL 8.0 (relational database)
- MySQL 5.0 (relational database)
- Objectivity 9.1 (object-oriented database)
- DB4O 5.0 (object-oriented database)
For each relational database, the appropriate object-relational mappings were applied separately with results gathered for each. To cater for the performance hit sustained by executing the test scaffolding code, a Normal run was also performed for each test to measure the time taken without any object store present. Analysis of the most interesting test results is summarised below, for results of all the tests run, please see section 11.2 of the Appendix.

### 2.1.6.1 Small Object Tests

The following classes were used to test each product’s capabilities for storing graphs of many small objects. Each object generated from these classes was filled with data which, by itself, should take up less than 1 kB of memory.

![UML diagram of classes used for the small object tests](image)

**Figure 2-14** UML diagram of classes used for the small object tests

### 2.1.6.2 Insert Performance

The aim of this test was to measure the time taken to insert small objects into each persistent store and how this performance changed with the number of objects persisted. The number of objects inserted was varied, and the time taken to insert each set of objects was recorded and graphed.

![Graph of insert performance for small objects](image)

**Figure 2-15** Graph of insert performance for small objects
Serialization fared well for small objects, offering performance execution times between PostgreSQL and MySQL. However, using basic serialization requires all existing objects to be read and rewritten on every insert, giving an expected $O(n^2)$ performance for $n$ objects:

$$E_{\text{serialization}}(n) = \frac{(R + W)n(n + 1)}{2}$$

Although the object-oriented databases were expected to fare better than the relational databases due to the lack of a mapping layer, DB4O and Objectivity were in the slowest three persistence strategies with execution 20 times slower than some object relational mappings. This may be due to the poor transaction rates offered by object-oriented databases and their reliance on caching to improve retrieval performance, which causes a negative impact on insert performance [Srinivasan and Chang 97].

For the relational mappings, foreign key aggregation produced the slowest times, which is to be expected since each object inserted causes a row to be inserted into each of the 5 tables for the classes involved in the object graph, plus the cost of reading the key value assigned to the referenced rows. Basic mapping and single table aggregation produced very similar results, since both need only insert a row into a single table for each object persisted.

$$E_{\text{foreignkeyaggregation}}(n) = (5W + 4K)n$$
$$E_{\text{basicmapping}}(n) = (W + 2S)n$$
$$E_{\text{singletableaggregation}}(n) = Wn$$

2.1.6.3 Select Performance For Single Objects

This test evaluates the execution time of each persistent store when accessing a specific object from an ordered set of 10,000. The index of the selected object (indicating its position in each store) was varied to see how the performance of each store might change when accessing different objects throughout the set.

This test produced some unexpected results (shown in Figure 2-16), since all stores except DB4O exhibited a large increase in execution time once the selected object’s index exceeded 7000. This could be due to those data stores caching the first 7000 rows in memory with subsequent accesses requiring a page to be loaded from disk. However, this wouldn’t explain the results for serialization, which also showed an increase past the 7000 point despite the fact that all objects are deserialized no matter which object is accessed and so the performance is expected to be constant. A possible explanation that encompasses this would be caching of recently-accessed disk pages implemented in the operating system which somehow DB4O does not take advantage of.
2.1.6.4 Large Object Tests

The following classes were used to test the performance of each product when storing graphs of many large objects. Large objects were generated using image and text data, with each object requiring around 150 kB of memory.

```
<table>
<thead>
<tr>
<th>Article id: int</th>
</tr>
</thead>
<tbody>
<tr>
<td>image: byte[]</td>
</tr>
<tr>
<td>title: String</td>
</tr>
<tr>
<td>content: String</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Writer firstName: String</th>
</tr>
</thead>
<tbody>
<tr>
<td>lastName: String</td>
</tr>
<tr>
<td>photo: byte[]</td>
</tr>
</tbody>
</table>
```

2.1.6.5 Insert Performance

This test measured the time taken to insert large objects into each persistent store. The number of objects inserted was varied in an attempt to reveal the order of complexity of the operation.
Unlike for small objects, serialization did not perform better than the other stores, but instead rose sharply from an initial execution time of 30 seconds for 100 objects to over 500 seconds for 300 objects, at which point it threw a memory exception. DB4O also performed badly and produced memory errors past 800 objects (not shown). The other stores exhibited a more exaggerated version of their behaviour for small objects, with the relational databases still outperforming the object-oriented ones and PostgreSQL producing the best results overall.

2.1.6.6 Select Performance For Large Fields

This test aimed to show the performance of each persistent store when selecting large data fields. The largest field from a single large object was requested from each store. The index of the requested object was varied and the execution time for each index was graphed. Serialization and DB4O were omitted from this test due to memory errors when initialising them with the test data.

Figure 2-19 shows the result of this test. Whereas for small objects the performance of MySQL was similar to that of the other stores, for large objects it shows a much steeper rise in execution time as the object index increases. The reasons for this are not obvious, although in similar tests on smaller objects all stores have been found to suffer performance loss at particular indices, which is probably due to paging. For this test, Objectivity performs substantially better than MySQL for large object indices; however the rise in execution time once the index surpasses 500 indicates that it may perform badly for higher indices that weren’t tested. PostgreSQL still performs better than all other stores, with even the worst mapping exhibiting approximately a third of the execution time of its nearest rival for all but the zero index.
2.1.6.7 Class Hierarchy Tests

A number of tests were run on objects that are part of a class hierarchy whereby each class adds its own member variables. All objects in Java are part of a class hierarchy, but often the superclasses don’t declare any member variables, and therefore do not affect the data storage requirements of the object. In the following tests, ‘object from a class hierarchy’ should be taken to mean that the object belongs to a hierarchy in which each class contributes to the object’s data storage requirements. The following hierarchy was used to generate the objects used in these tests:

```
Person
  firstName: String
  lastName: String
  age: int

Staff
  pay: int

OverseasStudent
  country: String

HomeStudent
  lea: String

Student
  year: int
```

Figure 2-20 UML diagram of classes used in the class hierarchy tests
2.1.6.8 Select Performance Test

This test aimed to see how each of the persistent stores would perform when selecting objects from a class hierarchy based on a conditional test across fields of a subclass. The test retrieved the firstName and lastName fields of all HomeStudents whose lea value was “Bristol”. The number of objects requested was varied and the execution time was recorded for each store.

![Figure 2-21 Graph of select performance for multiple class hierarchy objects from 10,000](image)

In these results, the table per hierarchy and table per path mappings exhibit very similar performance, which is to be expected since both only need to read from a single table to perform both the conditional test and fetch the required fields. Table per class performs worse, with a 50% slowdown on average when compared to the other mappings. This is likely to have been caused by it joining the subclass table on which the conditional test must be run and the superclass table that contains the fields.

2.1.6.9 Memory Usage Test

This test attempted to determine the secondary store usage of each of the persistent stores when persisting objects from a class hierarchy. 10,000 objects were generated and stored and the sizes of each database file were recorded and graphed.

In this test, as shown in Figure 2-22, Objectivity used an unexpectedly large amount of storage space to persist the given objects. It’s unclear as to why this should be, but a guess would be that it allocates buffer space so that fields with dynamic memory requirements such as Strings or arrays can grow and be stored contiguously. Most of the other stores use very similar amounts of storage space, with serialization performing the best since by design it provides the most compact representation of the object data. Out of the relational mappings, table per class performs the worst due to the extra overhead of using separate tables for the data of each class.
2.1.6.10 Conclusions

This section took a brief look at some of the performance issues associated with persistent stores. The widely varying results obtained indicate that this is not a trivial matter; slight alterations in the way that data is inserted or accessed can have major implications on performance, both in terms of execution time and memory usage. The generally poor execution times of the object-oriented databases also show that, despite the expectations that they would become the most efficient of the three strategies for storing objects, the sheer speed of relational databases manages to offset any performance hit suffered due to the mapping layer. Unfortunately, examining the performance of each object relational mapping in detail is beyond the scope of this report, but further experimentation should be done to come up with a set of heuristics for choosing the best mapping based on data usage statistics.

2.2 Persistence Layers

As highlighted in section 0, relational databases can offer big performance gains if used correctly, but the developer must first overcome the impedance mismatch between the object and relational schema. This section compares a number of existing products and designs that attempt to alleviate this problem by providing a persistence layer; a library that mediates between the custom application code and the relational database.
2.2.1 Java Database Connectivity (JDBC), ActiveX Data Objects (ADO)

Sun Microsystems and Microsoft have both developed libraries for Java and .NET respectively which attempt to provide a standard interface to relational databases. JDBC and ADO do not themselves map objects to relational databases, but instead provide tools to allow developers to access and update relational databases independent of the actual database implementation used. This gives the developer full control over how objects should be mapped to table schema and can be used to generate an object-oriented interface to existing legacy data. Typically applications that use JDBC and ADO have their queries hard-coded to store and retrieve only the required data from the database.

```csharp
SqlConnection conn = new SqlConnection("Data Source=(local);Initial Catalog=Northwind;Integrated Security=SSPI");
SqlDataReader rdr = null;
conn.Open();
SqlCommand cmd = new SqlCommand("select * from Customers", conn);
rdr = cmd.ExecuteReader();
while (rdr.Read()) {
    Console.WriteLine(rdr[0]);
}
```

**Figure 2-23 Example source code using ADO.NET in C#**

**Advantages:**

- Gives access to a large number of features via SQL.
- Allows developers to create custom mappings tailored to the data requirements.
- Dynamic queries can be built up through String manipulation.
- Potentially very efficient if queries are written to exactly match data usage.

**Disadvantages:**

- Custom code needed to persist each different class.
- Fragile; changes to data usage require changes to queries.
- Not entirely database-independent; some DBMSs interpret the SQL code differently to others.
- Efficient queries tend to be tightly coupled to the code that uses their results.

2.2.2 iBatis SQL Maps

*iBatis SQL Maps* provide an extra layer on top of JDBC and ADO which allows developers to separate their mapping code from their application code. The mapping between a class and a table schema is specified in an XML descriptor file, which the iBatis library reads and uses to generate the executed SQL statements. This allows the application code to be completely ignorant of the mapping used; instead it can simply request and use data objects in a similar manner to using an object-oriented database.

However, this decoupling of data retrieval and update from data usage can cause a drop in performance. Unlike using JDBC and ADO directly, where queries could easily be tailored to the specific data usage, in iBatis queries are intentionally separated from the application code that uses them. To gain the maximum efficiency from this approach, the developer would either have to write a separate query in the XML descriptor for each different usage or provide a single, highly parameterised query which would lead to a coupling that is almost as tight as using JDBC or ADO directly.
An example of this problem occurs when dealing with references to other objects. When an object is requested, all referenced objects can either be retrieved immediately (eager loading) or specifically requested by separate queries when the reference is traversed (lazy loading). The former could cause entire object graphs to be retrieved unnecessarily if the references are never traversed by the application code, while the latter could cause many separate queries to be executed if reference traversal is a common case.

Advantages:

- Separates out data access/retrieval code from application code.
- Object-relational mapping can be changed by altering just the XML descriptor file.
- Caching mechanisms are provided which can reduce the performance hit of executing multiple queries over the same objects.

Disadvantages:

- Tension between clean interfaces and good performance.
- XML descriptor file is not as easy to understand as pure JDBC or ADO code.
- Mapped classes are restricted to Map implementations or Java Beans in Java and IDictionary implementations or property classes in .NET.

2.2.3 Enterprise Java Beans 2

The Enterprise Java Beans standard was first released in 2000 by Sun Microsystems in an attempt to regulate the large numbers of ad hoc persistence mechanisms that were being implemented in Java. This section discusses the current 2.1 release of the specification [Sun 03], although version 3.0 is currently awaiting public approval and is addressed in section 2.2.6.
The EJB 2.0 specification advocates a separation between the business objects to be persisted, termed *Entity Beans*, and the operations that can be performed on them, which are defined in classes called *Session Beans*. Each bean can be accessed both locally and remotely, allowing large web applications to be distributed across multiple servers. To achieve this, the application developer must write a number of interfaces for each bean, as illustrated by Figure 2-27. Each bean also requires an XML file called a *deployment descriptor* that specifies various persistence and transaction properties.

The EJB specification offers two approaches to storing objects in a database; *bean-managed persistence* and *container-managed persistence*. The former essentially is the same as using JDBC directly; developers simply add methods to entity beans that are called when the bean’s data needs to be persisted or retrieved and these methods must contain the JDBC calls (or similar for non-relational persistence) for storing or retrieving the object. The latter leaves the persistence strategy up to the *EJB container*, which is typically the web application framework. Usually, this reads in the user-defined mappings and queries from the deployment descriptor then generates the necessary JDBC calls automatically. The differences between these two strategies are completely hidden from the bean’s clients.

**Advantages:**

- Container managed persistence automates the task of generating JDBC code.
- Bean managed persistence allows for custom mappings if necessary.
Disadvantages:

- Persistable classes are restricted to those that obey strict rules (e.g. have necessary interfaces and deployment descriptor).
- Very complex; it is easy for non-experts to create EJB applications that exhibit very poor performance.
- Design and levels of abstraction largely determined by specification and not by application domain.
- Large amounts of scaffolding across multiple files that must be kept synchronised and prevents quick prototyping.

A full discussion of Enterprise Java Beans is beyond the scope of this report; for more information please refer to the J2EE tutorial [Jackson et al. 05].

2.2.4 Java Data Objects (JDO)

Due to the growing complexity of Enterprise Java Beans and their close ties to relational databases, in 2002 Sun Microsystems proposed JSR-000012 [Russel 02] which put forward the idea of Java Data Objects (JDO). The aim of this was to provide a set of simple, storage-independent interfaces for persisting Java objects between JVM executions. This would allow developers to add persistence to their applications with minimal changes to the code and also allow for change between storage methods such as file I/O, relational databases and object-oriented databases without the need to recompile.

One of the main criticisms of the EJB 2 specification was that too many restrictions were places on persistable entity bean classes, severely restricting their usage as normal Java objects. Java Data Objects aimed to correct this by allowing developers to persist plain old Java objects, so that there was no difference between persistable and transient classes. However, to successfully persist these classes, additional information was required in the form of an XML descriptor file which allowed developers to specify metadata such as primary key fields, identity types[^3] and relationships for each class. An external tool, called the JDO Enhancer, would then read the descriptor file, alter the persistable classes so that they implement the PersistenceCapable interface and provide the necessary methods for storing instances of themselves.

[^3]: The identity type of a class refers to how the equivalence of two instances is determined. In JDO, this can be controlled by the data store, the application or each object can be seen as unique.
Balancing simplicity and efficiency in Web applications

Enhancer Input

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE jdo SYSTEM "...">
<jdo>
  <package name="uk.ac.ic.students">
    <class name="Student" identity-type="application">
      <field name="name" primary-key="true"/>
    </class>
  </package>
</jdo>
```

Student.jdo

Enhancer Output

```
PersistenceCapable
+ jdoGetPersistenceManager(): PersistenceManager
+ jdoReplaceStateManager(sm: StateManager): void
+ jdoProvideField(fieldNumber: int): void
+ jdoProvideFields(fieldNumbers: int[]): void
... READ_WRITE_OK: byte
... LOAD_REQUIRED: byte
... READ_OK: byte
... getters and setters...
```

```
PersistenceManagerFactory persistenceManagerFactory = new PersistenceManagerFactoryImpl();
PersistenceManager persistenceManager = persistenceManagerFactory.getPersistenceManager();
Transaction transaction = persistenceManager.currentTransaction();

// Create Student as normal, transient object
Student student = new Student();
student.setName("Marc Hull");
student.setCourse("Computing MEng");
student.setYear(4);

// Make student persistent by registering it with persistenceManager
transaction.begin();
persistenceManager.makePersistent(student);
transaction.commit();
```

Figure 2-29 Enhancing a class using an XML descriptor and the JDO Enhancer tool

In the application code, the JDO library provides a set of interfaces for persisting objects, retrieving objects and handling transactions. It is still up to the developer to decorate existing application code with calls to a transaction manager in order to ensure that the data store is not left in an inconsistent state.

```
PersistenceManagerFactory persistenceManagerFactory =
  new PersistenceManagerFactoryImpl();
PersistenceManager persistenceManager =
  persistenceManagerFactory.getPersistenceManager();
Transaction transaction = persistenceManager.currentTransaction();

// Create Student as normal, transient object
Student student = new Student();
student.setName("Marc Hull");
student.setCourse("Computing MEng");
student.setYear(4);

// Make student persistent by registering it with persistenceManager
transaction.begin();
persistenceManager.makePersistent(student);
transaction.commit();
```

Figure 2-30 Persisting an enhanced class using Java Data Objects

One of the big selling points of JDO is its ability to remove the need for SQL-like querying when retrieving persisted objects. This is done through *extents*, which provide a method for iterating through every instance of a particular class and its subclasses. However, this method on its own provides no means of filtering, ordering or aggregating the data, and implementing those tasks in the Java domain by iterating through the entire collection can be incredibly inefficient. Instead, for anything more than listing all instances of a certain class, a custom query language called *JDOQL* should be used. This
takes an extent and provides a set of methods for filtering down its contents in the domain of the data store.

```java
begin();
// Get the extent of the Student class
Extent extent = persistenceManager.getExtent(Student.class, false);

// Iterate over the objects and display them
Iterator i = extent.iterator();
while (i.hasNext()) {
    Student student = (Student)i.next();
    System.out.println("Student "+student.getName()+
        " (Year "+student.getYear()+")");
}
// Do some cleanup
extent.close();

Figure 2-31 Printing out a student listing using extents
```

The JDO Query Language contains a mixture of standard Java method calls and String expressions for filtering out the required objects. Unfortunately, since the filter expression itself is encoded as a String, there is no static type checking despite the fact that all the type information for the objects involved is readily available in their class definitions. There are also methods for moving variables from the Java domain into the query as parameters, however again their usage is not statically type checked.

```java
begin();
// Begin with the extent, which will fetch every student
Extent extent = persistenceManager.getExtent(Student.class, false);

// Define a filter String which will return all students who are on
// the courses in a courses list and in a year greater than
// minimumYear
String filter = "courses.contains(course) & year >= minimumYear";
Query query = persistenceManager.newQuery(extent, filter);

// Declare the variables we're going to feed in from the Java
// domain
String params = "List courses, Integer minimumYear";
q.declareParameters(params);

// Create our parameter objects
List courses = Arrays.asList(
    "Computing MEng",
    "Computing BEng",
    "ISE MEng"
);
Integer minimumYear = new Integer(2);

// Execute the query with the parameter objects
Collection emps = (Collection) q.execute(courses, minimumYear);

Figure 2-32 Displaying students from particular courses and years using JDOQL
```
Advantages:

- Near-orthogonal persistence; there is little difference between transient and persisted objects.
- Application code is independent of data store used.
- Developers can specify queries in Java code rather than SQL.

Disadvantages:

- Efficient queries still need to resort to a separate query language (JDOQL) which is not statically type checked
- An XML descriptor file is needed to add extra information to persistable classes.
- An enhancement step must be run on persistable classes between compilation and runtime.

2.2.5 Hibernate

Hibernate is an open source relational persistence system for Java\(^4\) which, like JDO, was written as a solution to some of the problems of EJBs. It allows ordinary Java objects to be stored in a relational database by using Java reflection and runtime class manipulation, eliminating the need for an extra enhancement phase between compilation and runtime. In a similar approach to iBatis SQL Maps (see section 2.2.2), the mapping between object and relational schema is specified in an XML descriptor file which is read by the Hibernate library at runtime and used to create the SQL statements to be executed.

```java
public class Event {
    private Long id;
    private String title;
    private Date date;

    public Event() {} // ... public getters and setters ...
}
```

**Figure 2-33 Example data class to be used with Hibernate**

```xml
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN" "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
    <class name="events.Event" table="EVENTS">
        <id name="id" column="EVENT_ID">
            <generator class="native"/>
        </id>
    </class>
</hibernate-mapping>
```

**Figure 2-34 XML descriptor file for mapping example class**

To retrieve objects from the database, a number of different approaches are offered, the most basic of which is to fetch a single object by passing its primary key to a `load(..)` method. An alternative is to query by example, whereby the developer creates an object of the type being requested, partially fills

---

\(^4\) A version of Hibernate for .NET, called NHibernate, is currently in development. See [http://www.nhibernate.org](http://www.nhibernate.org) for further information.
it with information and then asks Hibernate to fill in the blanks. This method can be used to fetch all objects of a particular type in which certain fields match particular values.

```
Cat cat = new Cat();
cat.setSex('F');
cat.setColor(Color.BLACK);
List results = session.createCriteria(Cat.class)
    .add(Example.create(cat))
    .list();
```

**Figure 2-35 Fetch all female black cats using Hibernate’s query by example mechanism**

More complex queries can be constructed by using the Criteria class, which essentially allows developers to build up an AST for an SQL-like query in Java code. Once a Criteria instance has been created it represents a query across all objects of a particular class. Specific columns can be filtered out by adding instances of the Projections class which accept field identifiers (the SELECT part of the query), rows can be filtered by adding Restriction instances that perform range and equality checks (the WHERE part of the query), while Order instances sort the results by a particular field (the ORDER BY part of the query). Although this ensures that the query itself is structurally sound at compile time, it still doesn’t provide type-safety on the query conditions, since object members are always referenced by name. This approach is also quite verbose and can make large queries difficult to read.

```
List results = session.createCriteria(Cat.class)
    .setProjection(Projections.projectionList()
        .add(Projections.rowCount(), "catCountByColor")
        .add(Projections.avg("weight"), "avgWeight")
        .add(Projections.max("weight"), "maxWeight")
        .add(Projections.groupProperty("color"), "color")
    )
    .add(Restrictions.disjunction()
        .add( age.isNull() )
        .add( age.eq(new Integer(0)) )
        .add( age.eq(new Integer(1)) )
        .add( age.eq(new Integer(2)) )
    )
    .add(Property.forName("name").in(new String[] { "Fritz", "Izi", "Pk" } ))
    .addOrder(Order.desc("catCountByColor"))
    .addOrder(Order.desc("avgWeight"))
    .list();
```

**Figure 2-36 Example of Hibernate’s criteria query mechanism**

Alternatively, queries can be built up in Strings by using the Hibernate Query Language (HQL), which looks like SQL but operates on the object schema rather than the relational schema. This means that the query language understands references, collections and inheritance, allowing complex object-based queries to be written much more concisely than their SQL counterparts.

```
SELECT count(cat) AS catCountByColor, avg(cat.weight) AS avgWeight,
    max(cat.weight) AS maxWeight
FROM Cat AS cat
WHERE (cat.name = 'Fritz' OR cat.name = 'Izi' OR cat.name = 'Pk')
AND (cat.age IN ('Fritz', 'Izi', 'Pk'))
GROUP BY cat.color
ORDER BY catCountByColor, avgWeight
```

**Figure 2-37 HQL version of the criteria query from Figure 2-36**

Finally, Hibernate allows for native SQL queries to be executed, which allow developers to take full advantage of the particular RDBMS they are using, but sacrifices the database and mapping strategy independence offered by the other query mechanisms.
Unlike the other persistence systems mentioned so far (with the exception of JDBC and ADO), Hibernate allows developers to customise the fetch strategy of their objects on both a global and a per-query basis. This is a partial solution to the lazy/eager loading problem (discussed in section 2.2.2), since each use of a query can specify on a per-field basis whether reference fields should be fetched immediately or only when needed.

Advantages:

- Near-orthogonal persistence; no need for an explicit enhancement phase, but XML descriptors and no-argument constructors are still required.
- Querying is performed on object schema and so is independent of the object-relational mapping and database used.
- Query by example is type-checked at compile time.
- Criteria queries are structurally checked at compile time.
- Developer can choose a fetch strategy suitable to the code that executes each query.

Disadvantages:

- Many different, custom query mechanisms that provide different advantages and disadvantages, making it unlikely for any one approach to be used solely throughout an application.
- Developers must know enough information about data usage at compile time to be able to optimise queries and choose the appropriate object-relational mappings.

2.2.6 Enterprise Java Beans 3

Due to the success of more lightweight persistence layers which impose fewer restrictions on persistable objects and require less scaffolding code than EJB 2, in 2004 Sun Microsystems began work on Enterprise Java Beans 3.0 which aimed to simplify the 2.1 specification while still providing the same features. The new specification [DeMichiel et al. 05] became available for public review in June 2005 and is expected to be approved during 2006.

The most obvious change in this version is the ability to persist ordinary Java objects without the need to hand-code the variety of interfaces needed in the previous version. Deployment descriptors have also been replaced by inline annotations, with many repeated settings being handled by global defaults so that they no longer have to be explicitly set in every bean.

With EJB 3.0, Sun Microsystems has successfully brought the specification up-to-date with many of the strategies used by Hibernate and JDO, but some of the problems associated with those technologies still remain. Entity bean developers must still have knowledge of how their classes are to be used in order to define efficient queries and mappings for them. Complex queries must still be written in a custom query language (EJBQL) that is not statically checked.

Advantages:

- Near-orthogonal persistence; persistable classes must be annotated.
- Querying is performed on object schema and so is independent of the object-relational mapping and database used.
Disadvantages:

- Queries must be implemented in a custom query language (EJBQL) that is not statically checked.
- Developers must know enough information about data usage at compile time to be able to optimise queries and choose the appropriate object-relational mappings.

## 2.3 Related Research

Although the topic of web applications has only emerged relatively recently, object persistence has been an active area of research for almost as long as object-oriented languages have existed. Originally, much work was done to provide database features in the Smalltalk language [Copeland and Maier 84] [Maier and Otis 86], then as language trends changed the emphasis switched to C++ and now Java. This section is a brief discussion of relevant research, both academic and commercial, and covers the current state-of-the-art in the field of object persistence.

### 2.3.1 PJava

During 1996, a team at Glasgow University worked on a tool called *PJava*, which attempted to provide orthogonal persistence for Java programs. In their paper [Atkinson et al. 96], they define three core design principles for their project to satisfy:

Orthogonal persistence is stated as the provision of persistence for all data irrespective of their type.

Transitive persistence specifies that an object’s lifetime is determined by its reachability, and so that any objects reachable from a persisted object must also be persisted.

Persistence independence requires that code operating on persistent objects is indistinguishable from code operating on transient objects, enabling software reuse between persistent and transient components.

Unlike the commercial products mentioned in section 2.2, PJava offers genuinely orthogonal persistence with no alteration needed to the classes which it persists. The developer specifies which transient objects should be persisted by using a simple interface that allows objects to be named in a global store (these objects are termed persistent roots).

```java
public class SaveSpag {
    public static void main(String[] args) {
        Spaghetti sp1 = new Spaghetti(27);
        Spaghetti sp2 = new Spaghetti(5);
        sp1.add("Pesto");
        sp1.add("Pepper");
        sp2.add("Quattro Fromaggio");
        try {
            PJavaStore pjs = PJavaStore.getStore();
            pjs.newPRoot("Spag1", sp1);
        } catch (PJSEException e) {
            e.printStackTrace();
        }
    }
}
```

*Figure 2-38 Code example for creating a persistent root in PJava*
Balancing simplicity and efficiency in Web applications

To retrieve a persisted object, the global store can be queried using the name supplied when the object was originally stored, which will move the object into an in-memory cache. Objects in this cache appear as normal Java objects to the rest of the JVM, but they have the special property that they can reference persisted objects that are only faulted in when the reference is traversed, essentially providing a lazy-loading implementation. Another feature of this system is that the Java classes associated with each persisted object instance are also persisted, including all method code, to ensure that the exact same object will be recreated when it is retrieved from the store.

The original PJava paper does not tie the project to any particular persistent store, instead providing an interface and framework for implementing a persistence layer. A prototype of the system, PJava0, was built using standard file IO and the results of testing it in real-world applications were published in [Jordan 96]. This study highlighted a few problems with the system; firstly the object cache and automatic faulting mechanism were implemented by modifying the Java Virtual Machine, so that applications written to use PJava were tied to a particular JVM implementation. Secondly, the decision to persist class code with objects provided interesting problems with schema evolution, since alterations to class code (during development or to fix bugs in a production system) would not affect objects that had already been persisted. It should also be noted that the lazy loading technique used can provide very bad performance when traversing many object references.

Overall, the PJava project is an interesting proof of concept. Many of its features can be found in current object-oriented database implementations and its emphasis on minimal impact to the application source code should be strongly considered for this project.

2.3.2 CDuce

Many existing persistence layers attempt to make persistence code more manageable by hiding it from the developer. CDuce takes a different approach by providing a language that had been designed with data storage and retrieval in mind. It offers a functional language based upon ML that has first-class support for reading and writing persistent data in the form of XML documents. It allows developers to declare types using XML structures, so that queries over XML documents can be expressed using language constructs that are checked for both structural correctness and type safety at compile time.

```plaintext
\[
\text{type ParentBook = \{parentbook\}[Person\*]}
\text{type Person = FPerson | MPerson}
\text{type FPerson = \{person gender="F"\}[ Name Children (Tel | Email)*]}
\text{type MPerson = \{person gender="M"\}[ Name Children (Tel | Email)*]}
\text{type Name = \{name\}[PCDATA]}
\text{type Children = \{children\}[Person\*]}
\text{type Tel = \{tel kind="home"|"work"\}[0'--'9'+] | '-'? '0'--'9'+]
\text{type Echar = 'a'--'z' | 'A'--'Z' | '_' | '0'--'9'}
\text{type Email= \{email\}[ Echar+ ('.' Echar+)* '@' Echar+ ('.' Echar+)+ ]}
\]

Figure 2-39 An example mapping between types and XML in CDuce

The main aim of CDuce is to provide an alternative XML transformation language to XSLT style-sheets. Transformations are performed by using pattern matching based on the mapped XML types and regular expressions then applying higher order functions to the matched data.

```plaintext
\[
\text{let names (ParentBook -> [Name\*])}
\text{<parentbook>x -> (map x with <person ..>{ n _\*} -> n)}
\]

Figure 2-40 CDuce query for obtaining a list of book names

CDuce shows that integrating querying and transformations into a language can produce very concise programs that provide many of the features that enterprise persistence layers lack, such as queries that can be both highly complex and type-safe. However, the language itself is tailored to perform a very specific task, whereas for web applications data retrieval is only a small part of a bigger system.
2.3.3 C# 3.0

Whereas CDuce is an example of a language constructed for the purposes of data querying and transformation, Microsoft hopes to add features to its established C# language in order to provide similar functionality. The C# 3.0 specification outlines a set of new features intended to provide a common interface for accessing data regardless of its source [Microsoft 05].

Firstly, lambda expressions have been added, which are implicitly typed anonymous functions and provide C# with some of the features of a functional programming language. Next, an SQL-like query language has been provided in the C# syntax and provides a common, statically-checked interface for querying collections, relational databases, XML documents and can be extended to cover additional stores.

The execution semantics of these queries is not specified exactly, but instead each query is translated into a chain of method calls and lambda expressions that are invoked upon the object provided in the `FROM` clause of the query. Any object that implements the required methods can therefore be queried in this way. For accessing a relational database, the passed lambda expressions can be inspected as expression trees and converted into the corresponding clauses of the SQL statement to be executed.

Overall, by providing a query mechanism that is used for both transient and persistent data, developers can more easily specify the data they want rather than how it should be collected, leaving the underlying implementation to determine the optimal method for retrieving it. It also allows for queries to be structurally checked at compile time, but does not ensure type safety. This approach provides some improvements over the querying mechanisms provided by existing persistence stores, but it still burdens the developer with the task of moving as much processing as possible into the queries in order to improve efficiency.

2.3.4 Haskell List Comprehensions

A project at Imperial College during 1996 covered the possibility of extending the Haskell functional programming language to query relational databases using its existing list comprehension system [Field and Hutton 96]. The project’s implementation translated certain list comprehensions into SQL queries, allowing the developer to easily move processing between the Haskell and database domains by moving code into and out of the body of the comprehensions. The paper contains results for tests with a varying proportion of the processing being performed in each domain to highlight the importance of using database processing to reduce both execution time and heap usage.

```
<expression> | <qualifier>, ..., <qualifier>
```

*Figure 2-41 General structure of Haskell list comprehensions*

The prototype implementation considers a subset of the full list comprehension language that can be easily mapped to SQL queries. The expression part of the comprehension is restricted to tuples of projector calls, where each projector takes a row of data and returns a single field, thus representing the `SELECT` part of the SQL query. The qualifiers can be either generators, which map a row from a relational table to an identifier, or filters, which are conditional statements that decide which elements are included in the returned list, representing the `FROM` and `WHERE` clauses of the SQL statement respectively.

```
(surname e, name p) | e <- employees, p <- projects, projnum e == num p, age e /= 40, budget p >= 1000
```

*Figure 2-42 Example list comprehension*
Balancing simplicity and efficiency in Web applications

```sql
SELECT employees.surname, projects.name
FROM projects, employees
WHERE employees.projnum = projects.num
AND employees.age != 40
AND projects.budget >= 1000
```

*Figure 2-43 Translation of list comprehension from Figure 2-42 into SQL*

Unfortunately, conversion of function calls that appear in the list comprehension are done based on prior knowledge of the semantics of the function, limiting the Haskell code that can be moved into the SQL domain. A more ambitious implementation would be to allow arbitrary functions to be included within the list comprehension, then to analyse the code of these functions to determine whether they can be converted into SQL or, if not, executed in the Haskell domain after the relevant data has been retrieved from the database. This would essentially automate the process of deciding which processing can be performed on the database by finding the largest amount of code that can be converted to SQL.

### 2.3.5 Nestor

Another Imperial College project from 2004 called *Nestor* ([Yeganeh 04]) concentrated on improving query performance by optimising JDBC calls. The project report highlighted fifteen commonly missed optimisation opportunities with JDBC calls and featured the results of a series of tests to highlight the speedup gained by performing each. The aim of Nestor was to detect these optimisation opportunities in application code and either apply them automatically or advise the developer of its findings.

The following optimisations mentioned are relevant to the persistence layer of this project and should be considered for the implementation:

- Connection pooling, to reuse connections that have been previously closed and avoid the cost of creating a new connection for each transaction.
- Using column indices instead of column names where possible to avoid the need for an extra lookup stage.
- Using `PreparedStatement` when the same query structure is executed multiple times with different parameters in order to avoid the overhead of sending the entire query request each time.
- Ensure that the correct column types are being requested from returned `ResultSet` s to eliminate manual conversion code.

Additionally, the web application framework developed as part of this project could attempt to detect and apply the following optimisations automatically in order to improve query performance:

- Reducing redundant data transfer by altering `SELECT *` queries to explicitly state only the columns required by the application code.
- Adding indices to columns that commonly appear in the `WHERE` clause of queries.
- Aggregating fine-grained queries across the same table into fewer, coarse-grained queries.

In Nestor, the `SELECT *` optimisation was automated by using the `Soot` compiler framework to detect the usages of the `ResultSet` instance returned by each query. This analysis was done statically, so was restricted to the method in which the query was executed and a limited subset of called methods. The usage of the `ResultSet` could not be traced into polymorphic method calls or into the invoking method since these would only be known at runtime.

For this project, data flow analysis could be used to infer an efficient SQL query from the application code that uses the data. However, since `ResultSets` should not leave the persistence layer, the analysis would have to be done on the usage of the persisted objects themselves, which could potentially be any class if the layer implements true orthogonal persistence.
Unfortunately the Nestor prototype implementation does not cover the optimisations involving automatic indexing of commonly used columns nor the aggregation of fine-grained queries across the same table. For this project, the former could probably be implemented through simple statistics gathering in the persistence layer, while the latter could use data flow analysis to detect loops containing reference traversal from persistent objects and perform a single query to prefetch the referenced objects.

Overall, the Nestor project provides a lot of useful information about possible approaches that this project could take to improve query performance. The results gathered from the use of Nestor in a real-life application demonstrate that these optimisations can have a large impact on the response time of a product.

## 2.4 Presentation Layers

The presentation layer of a web application is concerned with receiving page requests, invoking the necessary functionality from the business layer, gathering any content to be output and then returning the content to the user in the relevant format. Web application frameworks often provide tools or interfaces to help manage the intermix of content generation code and formatting code that occurs in this layer. This section briefly discusses a number of these tools and suggests possible improvements that this project could make to them.

### 2.4.1 HTML and CSS

The standard format for web pages has long been *HTML*, but the standard itself has changed regularly and is likely to continue to do so as the requirements of pages continue to expand beyond the language’s original purpose. Recently, more emphasis has been placed on customising the look of each web page for different media, such as for printing or display on mobile devices. This movement has given birth to *Cascading Style Sheets (CSS)*, which intend to allow the style of each page to be factored out such that it is separate from the content. Ideally, this allows for the HTML files themselves, which should contain only the content of the page, to be quite small, while the CSS files, which contain the styling for each type of supported media, only need to be loaded once per site.

However, the CSS standards regularly fail to fulfil their promises, mainly due to their ties to HTML which isn’t a suitable language for expressing content, but also because of differences between browser implementations. Although style sheets go some way to separating out the look of a page, they are still restricted by the layout of elements within the HTML itself, making it difficult to drastically change the page’s appearance. Often, many web applications make use of CSS to decrease page size and achieve graphical effects that are difficult in HTML alone, but also have an additional server-side layer that alters style on a per-request basis.

### 2.4.2 Java Server Pages (JSPs)

*Java Server Pages* provide a concise way of combining both static and dynamically-generated XML markup using Java. JSP files are essentially XML documents with special tags that handle control flow and allow the developer to escape into the full Java language to generate dynamic content. Before execution, JSP files go through a translation and compilation phase, where they are transformed into Java servlet classes.
Balancing simplicity and efficiency in Web applications

```html
<html>
<head>
  <\%= int localVariable = 1; %>
<table>
  <\%= "expanded inline data " + 1 %></td></tr>
</table>
</head>
...<\/html>

Figure 2-44 Example code fragment from a JSP page

```java
javax.servlet.jsp.JspWriter out = pageContext.getOut();
...
out.print( "<html>
" );
out.print( "<head>
" );
...
int localVariable = 1;
...
out.print( "<table>
" );
out.print( "   <tr><td>");
out.print( toStringOrBlank( "expanded inline data " + 1 ) );
out.print( "   </td></tr>
" );
...

Figure 2-45 Example code fragment after translation to a Java servlet

One of the more powerful features of JSPs is that custom XML tags can be created for them and associated with Java, which can then apply transformations to the page content. This allows developers to write their own tags that alter control flow in a custom way, call methods in application classes or simply refactor commonly used constructions.

However, JSPs are prone to two problems. Firstly, developers often have difficulty deciding which parts of the application code should appear inline and which should be separated into normal classes. For complex output, this can lead to JSPs filled with application code that is difficult to read and debug, or Java classes that build up XML code using String concatenation. Secondly, JSPs are often tied to one particular format, so often multiple JSPs must be written for each format the content must be rendered in.

2.4.3 WebWork

To avoid the problems associated with JSPs, some web applications use an additional layer between content generation and formatting. WebWork provides this by enforcing a clear separation between the formatting of the JSP files, known as views, and the content generation, which is performed by separate classes known as actions. This allows the same content generation code to be used with multiple JSPs, reducing code duplication.

To help reduce the amount of custom code in the JSPs themselves, WebWork provides a set of tags for performing common operations such as iterating through collections or accessing data from the action classes, and developers are encouraged to use only these for dynamic content rather than inlining custom Java code. WebWork also provides features for managing user input, by examining any name-value pairs in the query string or post data and matching the names to setter methods in the action class. If such a match is found, the expected type of the setter is detected and the value is automatically converted to that type. Finally, the setter is called with the converted value just before the main part of the action class is executed, so that all the user data is available to it via getter methods.

By making the JSP files smaller, WebWork reduces the burden of having to write a separate file for different output formats. However, it also introduces an XML configuration file that contains the mapping between action classes and JSPs, which is an extra resource that must be updated whenever new pages are added. Its automatic conversion of request data is also limited to primitive types or Strings; complex objects such as Collections or Maps must still be manually handled.
2.4.4 SiteMesh

To handle the problem of separating content from formatting so that the same page can be displayed in multiple different formats, many web applications pass their pages through some additional transformation layer before they are returned to the user. One popular implementation of such a system is SiteMesh, which registers itself as a page filter within the presentation layer of J2EE and is then able to intercept HTML pages after their dynamic content has been generated by servlet classes. The HTML is then parsed and transformed as specified by a decorator page, which is an XML file containing tags for inserting elements from the original page.

SiteMesh has proved popular with many web development companies due to its ability to decorate existing HTML pages. Site-wide elements such as header and footer bars can be easily applied to all pages via the decorator page without needing to change each of the existing pages individually. However, this is also a disadvantage when writing new pages, since HTML is not an adequate language for expressing page content, so applying a consistent style across every page can require very large and cumbersome transformations.

2.4.5 XML and XSLT

The problem of separating content from style is in no way restricted to web content; it has been a major problem of typesetting for many years. The DocBook typesetting system provides a solution in the form of XML and XSLTs, whereby the content writer constructs an XML page containing a special markup for specifying the structure of the document, which is then transformed by an XSLT into the required format. This strategy may sound very similar to that of SiteMesh, but the main difference is that the XML DTD for describing the content has been designed with this purpose in mind, so is much easier to transform into different formats than HTML. This approach has been adapted to the domain of web applications and is currently used in the Resin application server.

The big advantage of this approach over the separation provided by HTML and CSS is that the transformation has complete control over the outputted page. If necessary, the HTML structure between two different formats (say, printed media and on-screen media) can be completely different, which is impossible in the current widespread implementation of CSS. However, some problems still exist with this strategy, mainly due to the XML style sheets themselves (XSL files). Since these are written in XML, they can be difficult to understand and alter, which in this system makes changing any part of the site design a more time consuming task. Also, unless the content DTD is fixed and cannot be extended by the developer, a balance must be found when writing new page components as to whether or not new content tags should be defined to make use of them or existing content tags should be transformed into them.

2.4.6 Tapestry

The Jakarta Tapestry project attempts to provide a JSP-like system for merging static and dynamic content, but with extra features to manage form and session data. Rather than using custom tags to denote dynamic content, Tapestry looks for additional attributes added to standard HTML tags, allowing the pages to still be previewed in a browser without any of the dynamic content being generated.

---

5 Resin is an open-source application server available from http://www.caucho.com/.
To respond to user input and store data in the session, developers must implement Java classes that are tied to each page via an XML descriptor file (much like in WebWork). In these classes, session data that must persist between multiple pages can be specified by marking a getter/setter pair with the @Persist annotation. Whenever the setter is called, Tapestry will automatically add the data to the session so that it can be retrieved by calling the corresponding getter even in subsequent page requests. Listener methods can also be declared that will be automatically invoked whenever certain page events occur, such as an onClick(...) handler for when the user clicks a particular link.

```java
@Persist
public abstract int getCounter();
public abstract void setCounter(int counter);

public void doClick()
{
    int counter = getCounter();
    counter++;
    setCounter(counter);
}
```

**Figure 2-46** Page class example showing session properties and an event listener method

![HTML code example](image)

**Figure 2-47** The HTML file that uses the page class in Figure 2-46

Tapestry also provides a lot of support for automating HTML forms. First, a value object must be created, containing member variables for each field of the form along with their corresponding getters and setters. Then, in the HTML page, each input field of the form should have attributes added to it to tie it to a particular member of the value object. When the page is loaded, Tapestry will automatically fill the form with the value object’s current data. Upon submission, the value object will automatically be updated with the submitted data by calling the corresponding setter methods.

Overall, Tapestry provides many features that are helpful in reducing the amount of scaffolding code needed to manage session data and form submission, and these techniques should be considered for the presentation layer of this project.

### 2.5 Conclusions

This section started by comparing different types of persistent store to determine which would be most suitable for persisting objects in a web application. The main points of each store are summarized in Figure 2-48.

Object-oriented databases and serialization were found to offer the most convenient interfaces for persisting objects, since they provide the developer with simple interfaces that accept and produce objects. Relational databases were found to benefit from a relatively high level of standardisation, offering both an API and a query language that would work across all major vendor implementations, preventing the technology lock-in suffered with the other approaches. With the object-relational mapping strategy it would be possible to provide the benefits of all these systems, but could compromise performance due to the need to convert data between the object and relational schema. A number of approaches for implementing this conversion were discussed, and the estimated relative performance of each is outlined in Figure 2-49.
To further determine the capabilities of each of the persistent stores, a series of performance tests were run on each and the results compared. Overall, the relational databases exhibited lower execution times than the object-oriented databases and showed very similar memory usage, indicating that the greater amounts of research that have gone into relational systems offsets the performance hit suffered in the mapping layer. Serialization performed reasonably well for small objects, but caused memory exceptions with larger objects due to its need to deserialize all stored objects for even simple operations. The PostgreSQL relational database consistently performed the best out of all the stores tested and so shall be the target store of this project from now on.

Having determined that relational databases are the most efficient stores for persisting objects, various persistence layers were evaluated to see how each overcomes the problems of impedance mismatch and providing orthogonal persistence. The following main points of each layer are summarized in Figure 2-51.

---

6 The ODMG standard is not widely supported.

7 The core SQL standard is supported, but later additions have yet to be adopted consistently.
Balancing simplicity and efficiency in Web applications

<table>
<thead>
<tr>
<th></th>
<th>Table Per Class</th>
<th>Table Per Path</th>
<th>Table Per Hierarchy</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Single object insert</strong></td>
<td><strong>W</strong></td>
<td><strong>W</strong></td>
<td><strong>W</strong></td>
</tr>
<tr>
<td><strong>Single object insert with n superclasses</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Single object select</strong></td>
<td><strong>R</strong></td>
<td><strong>R</strong></td>
<td><strong>R</strong></td>
</tr>
<tr>
<td><strong>Single object select with n superclasses</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Single object select from superclass with n possible subclasses</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Single object update</strong></td>
<td><strong>W</strong></td>
<td><strong>W</strong></td>
<td><strong>W</strong></td>
</tr>
<tr>
<td><strong>Single object update with n superclasses</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[W = \text{Row write}, R = \text{Row read}, J = \text{Table join}\]

**Figure 2-50** Estimated performance of various inheritance hierarchy mappings

<table>
<thead>
<tr>
<th></th>
<th>JDBC/ADO</th>
<th>iBatis</th>
<th>EJB 2.1</th>
<th>JDO</th>
<th>Hibernate</th>
<th>EJB 3.0</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Standard Query Language</strong></td>
<td>☑️ 8</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
</tr>
<tr>
<td><strong>Database Independent Queries</strong></td>
<td>☑️ 9</td>
<td>☑️</td>
<td>☑️</td>
<td>☑️</td>
<td>☑️</td>
<td>☑️</td>
</tr>
<tr>
<td><strong>Mapping Independent Queries</strong></td>
<td>☒</td>
<td>☑️</td>
<td>☑️</td>
<td>☑️</td>
<td>☑️</td>
<td>☑️</td>
</tr>
<tr>
<td><strong>Queries Type Checked At Compile Time</strong></td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☑️ 10</td>
<td>☒</td>
</tr>
<tr>
<td><strong>Queries Structurally Checked At Compile Time</strong></td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☑️ 11</td>
<td>☒</td>
</tr>
<tr>
<td><strong>Orthogonal Persistence</strong></td>
<td>☒</td>
<td>☑️ 12</td>
<td>☑️ 13</td>
<td>☑️ 14</td>
<td>☑️ 15</td>
<td></td>
</tr>
<tr>
<td><strong>Per-Query Fetch Strategy</strong></td>
<td>☑️</td>
<td>☑️</td>
<td>☑️</td>
<td>☑️</td>
<td>☑️</td>
<td>☑️</td>
</tr>
</tbody>
</table>

**Figure 2-51** Table summarising the features of each persistence layer

---

8 SQL is a standard that most RDBMS vendors implement a core subset of, but variations do exist.
9 Small differences exist between the way each database implementation interprets the SQL provided by JDBC.
10 Query by example offers a limited querying mechanism that is type checked at compile time.
11 Criteria queries offer a limited querying mechanism that is structurally checked at compile time.
12 An mapping must exist for each persistable class in the XML mapping file.
13 An XML descriptor file must exist for persistable classes and they must undergo an extra enhancement step after compilation.
14 An XML descriptor file must exist for persistable classes.
15 Persistable classes must be annotated.
This study highlighted a number of features that are consistently provided by each persistence layer, such as portability to different database implementations and the ability to alter the object-relational mapping strategies used without having to alter any queries. However, it also showed several features that are either missing or only partially implemented in every product examined. All of the object-relational mappers used their own system for representing and executing efficient queries, with Hibernate in particular offering three possible query mechanisms, each varying in complexity and features, giving the developer a lot to learn in order to make the most of the system.

Structural checking and type safety are two more features that are either absent or severely limited in each persistence layer. In every system, complex queries have to be built up in Strings that are then parsed at runtime, so any syntactic mistakes are not discovered until the code is executed. Finally, each persistence layer requires some form of class alteration before transient instances can be made persistent. This can restrict the design of the web application and render the persistence of classes for which the source code is not available impossible.

It is also worth noting the amount of development effort involved in using each of the persistence layers that were evaluated. All require the developer to state the object-relational mapping to be used in an XML descriptor file or by annotating each class. Since manually profiling a web application to determine its object usage and then using that data to specify a hybrid mapping strategy that provides the best performance is a very time consuming task, it is surprising that none of the products attempt to automate this process. Efficient querying of data is another area that can severely affect performance, but each of the layers analysed requires the developer to write queries that will provide the most efficient means of fetching the data required by the rest of the web application, which in a large, dynamic program can be an impossible task. No attempt seems to be made to improve upon this by calculating the optimal query from how the application code itself makes use of the persisted objects.

Several research projects were examined to see what work had been done in this area and determine the current state of the art. The PJava project provided a system that demonstrates true orthogonal persistence and performs all querying in Java code, ensuring that it is structurally sound and type safe at compile time. Reports of using a PJava prototype in production systems were very positive, but its reliance on a modified JVM and inefficiency due to the lazy loading strategy for fetching fields have prevented its widespread use. A project developed at Imperial had similar goals, but tried to provide persistence in the Haskell language using its list comprehension system. In this project, efficient queries were constructed by analysing the list comprehension and shifting processing from the Haskell domain to the database domain. Performance results obtained from doing this showed it to provide a big improvement in both execution time and memory usage, but the implementation did not attempt to infer SQL from user-defined function calls, limiting its usefulness.

Two attempts to provide persistence features as a core part of a programming language were discussed. The CDuce project has created a functional language that allows developers to easily parse and transform XML documents. This adds the structural checking and type safety features absent in many persistence layers and makes some attempt to optimise access to decrease execution time. However, the language is heavily tailored for performing XML transformations and is unlikely to be useful when writing web applications, for which data access is only part of a larger system. Microsoft’s research into this area attempts to avoid this problem by extending their established C# language with persistence features. They intend to integrate SQL-like querying into the language’s syntax and encourage its use for accessing data from a range of sources including collections and XML documents. These queries will be structurally checked at compile time, but they are still not type safe and do not provide more optimisation opportunities than existing querying techniques.

The Nestor project was examined to find out what optimisations could be applied to application code accessing a relational database. Several design patterns were identified that should be used when implementing this project and guidelines for improving the performance of web applications were highlighted. Unfortunately, the prototype implementation did not cover how these optimisations should be automated, but results from manually applying them showed that a large performance increase could be gained.
Finally, a number of presentation layers were discussed to see how existing products simplify the task of generating and formatting page content. Figure 2-52 outlines the main features of each of these products.

<table>
<thead>
<tr>
<th></th>
<th>HTML/CSS</th>
<th>JSPs</th>
<th>WebWork</th>
<th>SiteMesh</th>
<th>XML/XSLT</th>
<th>Tapestry</th>
</tr>
</thead>
<tbody>
<tr>
<td>Full separation of content and style</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>✔️</td>
<td>✔️</td>
<td>x</td>
</tr>
<tr>
<td>Content representation is appropriate</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>✔️</td>
<td>x</td>
</tr>
<tr>
<td>Content generated in application language</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>x</td>
<td>✔️</td>
</tr>
<tr>
<td>Style represented in domain-specific language</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>x</td>
<td>✔️</td>
</tr>
<tr>
<td>Automated parsing of request data</td>
<td>x</td>
<td>x</td>
<td>✔️</td>
<td>x</td>
<td>x</td>
<td>✔️</td>
</tr>
<tr>
<td>Automates control of session variables</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>✔️</td>
</tr>
</tbody>
</table>

**Figure 2-52 Table summarising the features of various presentation layers**

Although some of the presentation layers discussed can be used in conjunction with others, no combination successfully provides a satisfactory mechanism for separating content and style. Many rely on HTML as a format for representing content and do not allow styles to perform arbitrary transformations of its structure, limiting the differences allowed between decorated and undecorated content. Few allow for content and style to be generated in separate languages suitable to their particular domains, which would be a useful feature for development teams where the tasks of writing content generation and formatting code are given to different people. Finally, support for higher-level constructs to help control forms that need to pass data between requests was generally very poor, leaving substantial room for improvement.

If implemented properly, it should be possible for a presentation layer to display the same content in as diverse formats as PDF documents and Swing GUIs without needing to change the content generation code. This would also bring benefits to other areas of web application development, such as end-to-end testing, by allowing the test code to examine the content generated by each page rather than having to parse and sift through the formatted output. It should also be possible to represent common page components such as forms and pagination controls that may need to persist across multiple pages using high-level constructs that reduce the need for session control systems to be manually coded.

Overall, this section has evaluated at many existing products and research projects and identified a number of areas that should be improved upon. In the persistence layer, a system for providing efficient, orthogonal persistence with structurally checked and type safe queries is needed. An interface similar to that of PJava, which uses standard Java code for querying, but with the efficiency of manual mappings and queries possible with JDBC and ADO would be very beneficial. In the presentation layer, a system that fully separates content and style, allows both to be constructed in a suitable format and provides high-level control of common page components would be superior to many existing implementations.
3 Requirements Elicitation

The premise of this project is that the primary goal of a web application framework is to provide developers with a set of tools to help decrease development time, improve maintainability and sustain good performance within their web applications\(^\text{16}\). As highlighted in Section 2, many attempts have already been made to achieve this, yet no one standard has emerged above the rest and many existing products still exhibit major changes between each release. Many vendors accept that web application frameworks have a very important place in most businesses, but few seem to agree on how they can be implemented while providing all of the goals mentioned above.

This section explains the steps taken to pin down a set of requirements for a usable web application framework that improves upon existing products by moving closer to the core goals of improvements in development time, maintainability and performance. These requirements were obtained in a number of stages:

- A *case study* of an existing, industry-standard web application was analysed to identify general problems with web application frameworks.

- A *questionnaire* was made available for web developers to fill in, in order to consult the users of this product.

- A *prototype* was created with a minimal specification in order to determine the feasibility of the project and set its scope.

- A *constraint set* was constructed by analysing features of the Java language and virtual machine specification.

The remainder of this section describes the requirements gathered for each of these stages in Subsections 3.1 through to 3.4, which were then used to construct the full specification for the system given in Section 4.

\(^{16}\) For the purposes of this project, performance mainly refers to request service time and secondary store usage, however other issues addressed by web application frameworks may include RAM management and reducing usage of bandwidth.
3.1 Case Study

The aim of this case study was to analyse a web application used in industry and identify common antipatterns concerning the usage of the web application framework interface. The study was not concerned with issues specific to the web application itself, but was instead designed to highlight general problems that affect a variety of frameworks.

3.1.1 About The Web Application

The web application chosen is used by companies for both external sites, concerned with advertising and display of information, as well as internal sites, where it provides organisational tools and allows staff to share files in a secure environment. Its design is modular and dynamic, allowing new functionality to be written separately from the rest of the application and deployed only to the clients that require it. Unfortunately, for confidentiality reasons, the name of this application cannot be given, however it is currently used by a variety of large financial companies in the UK.

For persistent data, objects are stored in a relational database, but the database implementation used is determined by each client, so compatibility across a number of systems is important. Currently, this is achieved through the Enterprise Java Beans 2.1 standard (see section 2.2.3), but it is in the process of being ported to use the new 3.0 version of the specification.

For presentation, the product is mainly accessed through a variety of web browsers, so compliance with HTML standards is encouraged, but it also produces content in a number of other formats, including RSS and PDF. Currently this is done through Java Server Pages (see section 2.4.2) with a variety of additional tools such as WebWork and SiteMesh.

3.1.2 Test Strategy

Since the web application contains over 500,000 lines of Java code, I needed a way to narrow down my study so that I could concentrate solely on the code that was likely to cause problems due to the interface between the application and the J2EE framework. To do this, I refined my search to problems affecting data access and usage, defining a problematic area to be some path or set of paths through the application code whereby the database queries could be altered to achieve the same result with shorter retrieval time. These areas are generally characterised by having one of two properties:

**Fine query granularity** is where each query executed returns too small a subset of the data, so several queries must be run to fulfil the data requirements of the application code. This was detected by finding pages that cause a large number of queries to be executed on the same table.

**Coarse query granularity** is where each query executed returns too large a portion of the data, so that only a subset is needed by the application code. This can be detected by finding pages that cause very general queries to be executed and then examining the corresponding application code to determine the data usage.

---

17 A software antipattern is the natural counterpart to a traditional design pattern. It describes a commonly used, badly designed solution to a problem that causes a performance decrease.
These heuristics were applied to two of the modules in the web application and their results analysed. Sections 3.1.3 and 3.1.4 discuss the findings of these tests for each module and discuss possible performance improvements that could be applied based on these results. For a full explanation of how these tests were performed, see section 11.3 of the Appendix.

3.1.3 Announcements Module

The first and simplest module to be analysed was the announcements module, which allows users to post announcements on a message board that anyone within the same community\(^\text{18}\) can view.

3.1.3.1 Code Analysis

This module was profiled to find which queries were generated when the announcements index page was viewed. The query logs were then analysed to find the following opportunities for optimisation in the source code:

Field Selection

The announcements index page displays a summary of each announcement on the message board; it does not display the main content of the announcement nor many other minor fields. However, in EJB 2 there is no way to fetch partial objects, so for each announcement in the index, all fields associated with that announcement are fetched.

This could be improved through lazy loading of field data such that queries for retrieving the field values of an object are delayed until the values are required by the web application. This would eliminate all the data loaded unnecessarily by the persistence system, although at the expense of executing multiple queries.

Conversion To Data Objects

In EJB, entity objects are considered as live objects, in that altering fields on those objects will automatically cause the persisted copy of that object in the database to be updated with the changes. Since the persistence layer of a J2EE application may reside on a different machine to the presentation layer, all methods invoked on live objects are performed using Java Remote Method Invocation, adding a substantial overhead to each getter and setter call.

To improve upon this, developers are advised to use the Transfer Object design pattern [Sun 02], whereby a non-live object (referred to as a data object or value object) is constructed using the fields of the entity object before being passed out of the persistence layer so that the RMI overhead only occurs once. Unfortunately, there is no mechanism to tell the persistence layer to automatically generate transfer objects, so this task is left to the session bean which loops through all returned entities and calls their `getData()` method. This could be improved by eliminating the generation of entity objects all together for cases where only transfer objects are required.

3.1.3.2 Optimisation Results

To determine whether these optimisations would be worthwhile to implement, the service time for the announcements index page was profiled for varying numbers of announcements. The source code for

\(^\text{18}\) In this web application, user access is determined by communities (user groups).
the module was then manually modified to hand-optimise the queries executed and the test re-run on the optimised code. The following graph shows the speedup between the unoptimised and optimised versions.

![Graph showing speedup for announcements index page service times with optimised code](image)

**Figure 3-1** Speedup between unoptimised and optimised versions of the announcements module

This demonstrates that, on average, a 2 times speedup is possible for even this simple module by implementing more efficient queries that fetch only the required data from the database. For the full results of this test, please see section 11.3.1 of the Appendix.

### 3.1.4 Calendar Module

The calendar module allows users to add timed entries to a schedule that is shared by other members of the same community. It allows existing entries to be viewed grouped by day, month, week or year.

#### 3.1.4.1 Source Analysis

As with the announcements module, pages from this module were profiled to find the queries generated for them and these were then scanned for fine query granularity and coarse query granularity. Instances of each were then traced back to the source code from which they originated. A number of optimisation opportunities were found, many of which were variations on those found in the announcements module. However, one new type of optimisation was found to have a large impact on the number of queries generated:
Table Schema Optimisation

Although it is typical to represent database data in normal form, in some situations duplication of data may be preferential to avoid expensive JOIN operations for commonly accessed objects. In this test, each calendar entry required a separate SELECT statement to be executed, performing a JOIN between multiple tables. For large numbers of entries, these common JOIN operations were found to severely affect the performance of the web application. This could be improved by identifying commonly-accessed fields and inlining their values into the table of the referring object so that the JOIN can be eliminated.

3.1.4.2 Optimisation Results

To test these optimisations, they were manually applied to the module’s source code and the service times for the month view were profiled for varying numbers of calendar events. Figure 3-2 shows that substantial benefits can be gained by performing optimisations upon this module. The initial 3 times performance increase for no calendar events is probably due to the reduction in Java code executed, since the original code will still try to perform processing on the empty result set. However the subsequent increase, which levels out at an improvement of around 17 times, is most likely to have been caused by the elimination of JOINs as mentioned above.

![Speedup for calendar month view service times with optimised code](image-url)

*Figure 3-2 Speedup for calendar month view service times with optimised code*
3.2 Questionnaire

Part of my requirements elicitation consisted of determining the necessary steps for making a usable, developer-friendly product. To achieve this, I wanted to find out which tools and technologies are currently used by web application developers and which aspects of those technologies they considered to be most important. I constructed an online questionnaire which was targeted at web developers and gathered results over a four-week period. These results were then used to form and prioritise the requirements for the various components of this project.

This section contains a brief analysis of the most interesting results, from which this project’s requirements were formed. The full results can be found in section 11.4 of the Appendix.

3.2.1 Table And Query Optimisation

Two of the questions asked concerned whether the participant manually optimises table schema and queries for efficient data access. The aim of this question was to determine how important these factors are to developers when writing web applications, and hence the priority of addressing them in this project. As shown in Figure 3-3, a large proportion of those who responded (70%) said that they do optimise table schema, indicating that it is a major factor. An even larger number (75%) said that they optimise queries.

![Figure 3-3 Graphs of questionnaire results concerning table and query optimisation](image)

Both of these indicate that the automation of performance optimisation at the database level should be a priority of this project. In response to this, the requirements for the persistence layer were adjusted to make choosing the most applicable table schema for each class a priority. Many of the requirements concerning the automated application of query-based optimisations found in the case study tests were also made core requirements.
Another question asked whether each participant would still want to be able to manually affect how data is stored if an automatic persistence system was used. Although half of the respondents said that this was not important, a considerable number (40%) said that this would be necessary. Intermixing automatic and custom mappings is likely to be a difficult task, so it was added as an optional requirement for the persistence layer.

The comments added to this question revealed that some participants did not have faith in an automatic system being able to match the performance of manual mappings. A possible remedy to this is to make the mappings and ‘thought process’ of the automated system visible to the developer. A tool for visualising the mappings used was added as an optional requirement.

### 3.2.2 Separation Of Content And Formatting

The comparison of various presentation layers in the background section already highlighted the generally poor separation between content and formatting (see section 2.4), so a question was asked to find out whether this is a major issue in web application development. As shown in Figure 3-5, 65% of respondents said that their sites do need to offer content in multiple formats, indicating that the need to improve on existing content separation methods should be a main requirement of the presentation layer.
Do your websites need to offer their content in multiple different formats (i.e. HTML, RSS, WAP...etc.)?

- Yes, 13, 65%
- No, 7, 35%
- Don't know, 0, 0%

**Figure 3-5** Graph showing the proportions of respondents who need their websites to offer content in multiple formats

Two further questions probed the extent to which the separation between content and style should be provided. The majority of respondents said that they preferred not to mix graphical interface code (HTML, CSS) with content generation code, ruling out a number of existing presentation layers that allow dynamic code to be inserted inline into HTML documents. This implies that the content generation should be performed in application code with little or no mention of formatting. The ability to build up a representation of a page’s content in application code was added as a major requirement of the presentation layer.

When adding functionality to your sites, do you prefer to mix writing code to generate content with writing code for creating the graphical interface?

- Yes, 5%
- No, 70%
- Don't mind, 25%

**Figure 3-6** Graphs showing the preferred separation of concerns within the presentation layer

A fair number of respondents (45%) said that the responsibility of writing content code and presentation code was given to different people. In this case, it is likely that the people responsible for page presentation prefer to work in a presentation-specific language as opposed to the one used to write the main application. However, for those who work alone, it is more likely that they would prefer to use the application language for expressing page presentation. To cater for both, two methods of formatting page content could be provided. The first would apply transformations in the application code to convert the content tree into a formatted tree representing the document to be output. The second would associate a set of formatter templates with each content node. Before output, the
presentation layer would apply the correct formatter template for each node, substituting special variables in each template for the actual page content.

![Pie chart showing responses to a question about the importance of output specifications to web developers.](image)

**Figure 3-7** Graph showing the importance of output specifications to web developers

One question asked participants how important it is to ensure that their site’s pages are compliant with HTML or XHTML specifications. Since a majority of respondents (85%) responded that this was important to them, the ability to ensure or, at least, check that pages being produced by the site are compliant with a particular specification was added to the main requirements of the presentation layer.

### 3.2.3 Prioritising Of Requirements

Participants were asked to prioritise a number of different aspects of web application development based on their importance. Since the requirements of this project are quite numerous and varied, the results of this question were used to determine which of the requirements of this project should be compulsory and which should only be attempted after the core requirements have been achieved.

Analysing the results (shown in **Figure 3-8**) revealed that providing a consistent site design and ease of navigation was the most important aspect of a web application, so this remains as one of the core requirements of the presentation layer. Extensibility and maintainability was also rated highly, and is part of the core requirements for all the components of the project. Short page load times were the third most important aspect, and should be achieved by the big emphasis on performance in the requirements of the persistence layer. Thorough testing was, surprisingly, one of the least important aspects, so the provision of a testing framework in the presentation layer was moved to an optional requirement.
3.2.4 Target Language

Two questions were asked to determine the target language for the web application framework. Firstly, developers were asked if they prefer using an object-oriented approach to writing websites, to which a large number (60%) of respondents said they did. They were then asked if they prefer to use a statically-typed language when writing websites, to which half the respondents said they did with a further 20% saying that they didn’t mind. This indicates that the target language of the web application framework (or, at least, the language that it should support for web applications) should contain both of these features.

Figure 3-9 Graphs showing the language preferences of web application developers
Another question asked participants which technologies they had used to write websites. Out of the most popular responses, Java encourages both object-oriented design and is statically typed, so seems the most appropriate language for this project.

![Image of bar chart showing technologies used to write websites]

**Figure 3-10** Graph showing the technologies used to write websites

### 3.3 Prototype

In order to scope the requirements of this project and spot potential problems early on, a prototype was produced which aimed to implement enough of each component of the system to be able to demonstrate a complete round-trip, from request to persistence to presentation. The following specific requirements were given for the prototype as a whole:

- The server should receive web requests, lookup the necessary page class of the web application and invoke it.

- The web application should be able to store, retrieve and remove objects from a persistent collection. It is sufficient for retrieved objects to be eagerly loaded, and to be persisted using only one mapping strategy (foreign key aggregation for object graphs and the table per class mapping for class hierarchies is recommended).

- The web application’s page classes should be able to construct and return simple representations of content (i.e. a page with headings, paragraphs and form input elements).

- The server should use some default transformation to convert the page content into an HTML page that is returned to the user.

Development of the prototype raised a number of interesting problems that had to be overcome. These problems, along with possible solutions, are summarised in the following sections.
3.3.1 Semantics Of Using The Collection API

The prototype provides a new implementation of the Collection interface called PersistedCollection which automatically persists to the database any objects that are added to it. However, to truly provide persistence, instances of PersistedCollection from one execution of the application must be made available to other executions so that the persisted objects can be retrieved from them. This requires there to be a way of relating instances of PersistedCollection together between executions of a program so that objects added to one instance become available to all related instances. The following are some possible solutions to this problem:

A single, global collection could be provided. Any objects that are placed into any PersistedCollection instance immediately become available to all other PersistedCollection instances. This approach is very simple, but is not desirable from both a use and an implementation viewpoint. Firstly, it is likely that the developer will not be interested in all persisted objects when accessing a PersistedCollection, so extra code would be needed to iterate through the collection and find only the relevant objects. Secondly, objects of any class could be stored in this collection, with each object being stored in the table corresponding to its runtime type (and supertypes). Therefore, to fetch the whole collection of objects, multiple SELECTs with JOINs must be used to fetch all the object data from the database.

The developer explicitly states which PersistedCollections are related by instantiating them with String names. Any objects added to a collection with a certain name are accessible to all collections with the same name. This gives the developer complete control over separating persistent objects into logical collections based upon the design of the application. However, it is more inefficient than the global store approach since a separate table must be maintained for each different name, containing foreign key values to identify the objects belonging to that collection. When a collection is accessed, this table must be JOINed with the class tables that store the object data.

A compromise solution is where PersistedCollections are related by the least specific class which they can contain (i.e. their generic type). This allows the developer to automatically filter out objects by type by providing the required class as a parameter when instantiating the collection. It is also efficient to access, since objects are stored by their class, so the number of JOINs needed to access a collection is reduced.

The strategy chosen for the prototype was to relate collections by the class of objects they accept, providing an interface demonstrated in Figure 3-11. This is not the optimal solution, so additional work should be done in future iterations of the project to see if relating collections by name can be performed in an efficient way.

```java
// Process One
Collection col = new PersistedCollection(A.class);
col.clear();
System.out.println(col.size()); // Prints 0
A a = new A();
col.add(a); // Add an A to the collection

// Process Two
Collection col = new PersistedCollection(A.class);
System.out.println(col.size()); // Prints 1
```

Figure 3-11 Example of the persistence API provided by the prototype
3.3.2 Object Updates

In other persistence layers, such as Enterprise Java Beans, any changes made to persistent objects are automatically applied to the database (see section 2.2.3). However, as mentioned in the background research, this can cause unnecessary database queries when multiple members of an object are changed one after the other. Commonly, this is avoided by using the transfer object design pattern (mentioned in section 3.1.3.1), which requires a data or value class to be created that holds the entity data and allows changes upon that data to be performed without being committed back to the database. Objects in the database can then be updated efficiently by committing all the changes to the data or value object at once.

For this prototype, I decided to invert this design and provide copy semantics by default. When an object is persisted, it should not be considered a ‘live’ representation of the data in the database, but instead a copy of the data on which changes can be made that are not automatically committed. To alter an object’s data in the database, it must be passed back to the persisted collection as a parameter of the `update(..)` method. This means that the common case of several object members being updated together is performed efficiently by default, while also allowing the ‘live’ object representation to be simulated by making a call to `update(..)` inside each setter method of the persisted object.

```java
PersistedCollection col = new PersistedCollection(A.class);
col.clear();
A a1 = new A();
a1.f = "Hello, world!";
col.add(a);

a1.f = "Goodbye, world!";
A a2 = col.iterator.next();
System.out.println(a2.f); // Prints "Hello, world!"

col.update(a1);
System.out.println(a2.f); // Prints "Hello, world!"

a2 = col.iterator.next();
System.out.println(a2.f); // Prints "Goodbye, world!"
```

**Figure 3-12 Example of the copy semantics of PersistedCollection**

In order to implement this behaviour, there must be some notion of object identity so that the `update(..)` method knows which rows of the database correspond to the passed object enabling it to update them accordingly. Obviously, this cannot be based upon the values of an object’s members since they could have completely changed before the `update(..)` method was invoked. An object’s address (returned by `System.identityHashCode(..)` can also not be used since it may be used by another object if the in-memory representation of the first object is garbage collected. Instead, some unique identifier must be attached to each object in the database, which can then be mapped to the object’s address while it exists in memory.

In the prototype, this problem is solved by adding an object ID column to every table in the database and ensuring that the rows representing an individual object share the same, unique object ID value. When an object is retrieved from a `PersistedCollection`, a mapping between the object and its object ID is stored in a `WeakIdentityHashMap`. This map must perform lookups based on an object’s `identityHashCode` (i.e. memory address) rather than using its `equals(..)` method since this may be overridden to compare the object’s member values. It must also maintain only weak references to its keys and remove the corresponding object ID values when the key object is garbage collected. This is to ensure that the mapping does not prevent objects from being garbage collected which would cause a memory leak.

Unfortunately, with this system, there is no way to automatically determine which member values of the object have been altered, causing the `PersistedCollection` to update the entire object, including all referenced objects, whenever the `update(..)` method is called. An alternative,
potentially more efficient solution is to fetch the object from the database and compare each member value to find out which have changed. However, the cost of fetching the entire object graph is likely to negate much of the performance gained by not updating fields that haven’t changed. Many existing persistence layers solve this problem by intercepting setter calls and marking the corresponding field as dirty, then only updating dirty fields when the object is updated. However, this cannot be applied to members that are accessed directly. More work should be done on this to find further solutions to this problem.

### 3.3.3 Database Garbage Collection

When an object is persisted by being added to a `PersistedCollection`, all referenced objects must also be implicitly persisted to ensure that the complete object may be rebuilt when retrieved from the collection. However, since `PersistedCollections` contain all persisted objects of a particular class, a decision must be made as to whether implicitly persisted objects should be directly accessible from the `PersistedCollections` relevant to their classes.

Making implicitly persisted objects directly accessible may be confusing to the developer, especially if they were persisted from private members that would not normally be visible. It would also allow them to be individually removed from the collection, in which case the original object that referenced them cannot be fully reconstructed.

```java
class A {
    private B b;
    public A() {
        b = new B();
    }
}
PersistedCollection colB = new PersistedCollection(B.class);
colB.clear();
System.out.println(colB.size()); // Prints 0
PersistedCollection colA = new PersistedCollection(A.class);
colA.clear();
A a = new A(); // Private member b is assigned in constructor
colA.add(a); // B is implicitly persisted because it is referenced by A
System.out.println(colA.size()); // Prints 1
System.out.println(colB.size()); // Prints 1
```

**Figure 3-13** Example code showing problems with directly accessible, implicitly persisted objects

However, if implicitly persisted objects are not directly accessible then it is possible to make persistent objects completely inaccessible, leading to a memory leak\(^{19}\). This is demonstrated by the example code in **Figure 3-14**, where an instance of the B class is implicitly persisted, but its reference is then replaced by a reference to a new B instance. In this case, the original B object would still exist in the database, but it wouldn’t be accessible from a `PersistedCollection` nor from the A object which originally caused it to be persisted.

---

\(^{19}\) This would be a memory leak in secondary storage, which may not seem as serious, but could cause unnecessarily large database files to be created over time in systems that generate lots of objects.
PersistedCollection col = new PersistedCollection(A.class);
A a = new A();
a.b = new B();
col.add(a);
a.b = new B();
col.update(a);

Figure 3-14 Example code showing problems with directly inaccessible, implicitly persisted objects

A potential solution to this problem is to delete the original B object when the A object is updated. This strategy assumes that it is possible to detect that the b reference has changed, which is discussed in section 3.3.2. It is also not always advisable, since there may be multiple references to the same B object which would now all be null. What is needed in this case is a form of garbage collection, so that objects are only removed from the database when they are no longer accessible from the application code. A simple solution would be to keep a count of the number of references to each implicitly persisted object which is decremented whenever a reference to it is lost, then to remove the object entirely when its reference count reaches zero. However, as with reference counting garbage collection in traditional programming languages, this method is flawed if cycles exist in the object graph. Further work should be done on this subject to find a workable solution for the final product.

3.4 Java and the JVM

This section does not intend to provide an introduction to the Java language, but instead gives a brief overview of some of the advanced features and restrictions imposed by the language and its runtime interpreter. For each of these features, the problems they pose on the persistence system are discussed and potential solutions highlighted.

3.4.1 Reference Semantics

All objects in Java are implicitly accessed through references to the object’s data on the heap, rather than direct addressing of the data itself. This allows the same object data to be referenced from multiple places, where each reference is referred to as an alias. Aliasing is a fundamental part of the Java language, allowing for cyclic runtime object graphs and leading to access or modification of data from a number of different places. To successfully provide orthogonal persistence of objects in the Java language, reference semantics must be supported such that aliasing is preserved when objects are stored and retrieved.

Typically, copy semantics are preferred in persistence systems since they are easier to implement and reflect the actual duplication of data that occurs. Under copy semantics, when an object is persisted its data is copied into the persistent store, but from then on the in-memory and persistent versions of the object exist separately from one another. Modifying the in-memory object does not affect the persisted data, which can be thought of as a snapshot of the object from the point when it was persisted.

Since the persisted data is entirely disconnected from the in-memory data, an object identifier (oid) number is often needed in order to refer to persisted data for a specific object. When an object is stored, an oid is returned which can then be used to retrieve, update or delete that object’s persistent representation. When the persisted object data is retrieved, an entirely new object is reconstructed using the persisted data, so it is possible for multiple copies of the object to be spawned from the same data and exist entirely independently.
Balancing simplicity and efficiency in Web applications

As Figure 3-15 shows, such a persistence system is far from transparent, forcing the user to keep track of object identifiers and update the persistent data when necessary. To offer a truly transparent persistence system, an additional layer must be constructed to connect the in-memory object data and the persisted representation. In particular, it should ensure that the following properties hold:

3.4.1.1 Storing an object and then fetching it produces the exact same object.

It is not sufficient to store an object and retrieve a copy of it with identical data. Storing an retrieving an object must produce a reference to the same object as the one in memory. In Java, references may be compared by '==' or System.identityHashCode(...) to distinguish the two cases.

```
A a1 = new A();
a1.s = "Hello, world!";
Oid o1 = store(a1);

a1.s = "Goodbye, world!";
A a2 = retrieve(o1);
System.out.println(a2.s); // Prints "Hello, world!"

a2.s = "Farewell, world!"
update(o1, a2);
System.out.println(a1.s); // Prints "Goodbye, world!"
```

Figure 3-15 - Java example demonstrating copy-semantics with store, retrieve and update persistence operations.

However, it is sufficient for any case where it is not possible to distinguish between the two objects to be allowed. For instance, if the example in Figure 3-16 was altered such that the in-memory data of a1 was garbage collected after it was stored but before a2 was retrieved, then we would have no means of verifying that a1 and a2 are identical, so in this case a2 may be a completely new object. This applies similarly to JVM crashes or normal program termination; there is no need to maintain references across JVM executions (in fact, this would often be impossible due to OS restrictions).

In essence the implementation should provide ‘observational equivalence’ between objects; that is, two objects are equal if there is no way of distinguishing them within the Java language. In practical terms, System.identityHashCode(...) allows the actual addresses of objects to be obtained and compared even after garbage collection or JVM termination, but these cases are unlikely to occur in real systems and so will not be considered further. This constraint may also be relaxed for immutable objects, since it is rare that these will be compared using pointer equivalence, and doing so allows immutable values, such as primitives, to not have their own oid value in many cases.

3.4.1.2 Updating the in-memory data of an object updates the persisted data.

Once an object has been stored, modifying that object’s data should cause the persisted data to be automatically updated. From Section 3.4.1.1 above, it is obvious that this will only be observable once the original data has been garbage collected, since otherwise retrieving the object will return the exact same object which will obviously reflect the changes. For testing purposes, a ‘disconnect’ operation may be used to separate an in-memory object from its persisted data. After disconnecting an object,
retrieving it will produce a copy containing the current persisted data, which can then be compared to the data of the original.

```java
A a1 = new A();
a1.s = "Hello, world!";
Oid o1 = store(a1);
a1.s = "Goodbye, world!";
disconnect(a1);
A a2 = retrieve(o1);
assert a1 != a2; // a1 has been 'disconnected'
assert a1.s.equals(a2.s); // a2 has the latest copy of a1’s data before disconnection
```

Figure 3-17 - A suitable test case for determining whether a stored object has had its persistent data updated.

### 3.4.2 Field Modifiers

Java contains a number of field modifiers that give special semantics to the fields that they annotate, which where possible should be preserved by any orthogonal persistence system. The following lists some modifiers that must be given special consideration during the persistence phase:

#### 3.4.2.1 Static

Static fields exist on a per-class basis rather than a per-instance basis. Two possible interpretations of this are possible; either a static field is persisted as an instance field or it is not persisted at all. If static fields are persisted, then retrieving an object with a static field would cause the current value of the static field to be overwritten. This is not desirable since static fields usually refer to some global state which should not persist between JVM executions, so it is preferable to omit such fields from persistence.

#### 3.4.2.2 Final

Final fields must be initialised to a value and cannot be altered. By definition, it is impossible to successfully persist these since they cannot be modified when a persisted object is retrieved. Therefore, they will be omitted from persistence.

#### 3.4.2.3 Transient

The transient modifier is used to denote that a field does not belong to an object’s persistent state, and should therefore be ignored by any persistence system.

### 3.4.3 Garbage Collection

Unlike many other languages, Java does not allow objects to be explicitly destroyed, but instead reclaims their memory when they become inaccessible. For a persistence system, recreating this behaviour is particularly difficult, since the persisted object graph could span a much larger volume of data and so determining reachability can be very time-consuming. However, forcing the user to explicitly destroy persisted objects is complicated by the fact that fields of objects can be references to other objects which, through aliasing, may also be referenced elsewhere. Figure 3-18 shows such a problem, highlighting that the user must disambiguate between whether the destruction of an object should cascade down into the objects it refers to. In other languages, this is often achieved through...
Balancing simplicity and efficiency in Web applications

adding destructor methods to a class, however for our purposes this is not an option, since it would
require the user to know in advance which classes are going to be persisted in order to add destructors
to them, which violates a fundamental property of orthogonal persistence.

```java
class A { int ai; public A(int ai) { this.ai = ai; } }
class B { A a; public B(A a) { this.a = a; } }
class C { A a; public C(A a) { this.a = a } }
B b = new B(new A(12));
C c = new C(b.a);
Oid o1 = store(c);
Oid o2 = store(b);
destroy(o1);
B b2 = fetch(o2);
System.out.println(b2.a.ai); // "12" or NullPointerException?
```

Figure 3-18 - Code example demonstrating the problems of explicit object destruction.

To provide garbage collection at the persistence level, a set of root objects must be identified from
which reachability of other objects is determined. The root objects must be the gateway from the Java
domain to the persistence domain, such that they are the only objects directly referenced from the Java
domain and all other accessible objects are referenced by paths from the root objects. From Section
3.4.1 above, it is obvious that this set of ‘gateway objects’ is those whose object identifiers are known
in the Java domain, since only then can those objects be fetched directly via a retrieve operation.
However, it is likely that in the final persistence system, object identifiers will be hidden from the user
and managed by an additional layer, which will need to maintain a list of gateway objects.

Once the root objects have been determined, any objects that are not reachable by traversing the
reference path from those roots are not accessible from the Java domain and can therefore be
reclaimed. In the persistence domain, a reference between two objects may be represented as a foreign
key constraint or by embedding the data of the referred object into the referring object’s table, but
either way the principle of reachability is the same. A number of algorithms have been devised for
determining reachability in the Java heap, so the following discusses potential modification of these
algorithms for determining reachability in a persistent store:

### 3.4.3.1 Reference Counting

It is possible to determine reachability in an acyclic object graph based on the number of references
pointing to that object. When a new reference is formed to the object, the count is incremented, and
when an object is deleted then the count of all objects it references is decremented. If the reference
count of an object reaches zero then there are no paths leading to it, so it may be deleted. In this way, a
single delete can cascade down a path of newly-unreachable objects, while those still accessible due to
aliasing will remain in-tact.

The main advantage of reference counting is that it is performed incrementally; reachability is
determined as references are created and removed without the need to traverse the entire object graph.
This is highly advantageous when the object graph is large and a full traversal is very time consuming,
which is likely to be the case in a persistent store. However, cycles in the object graph will cause
objects to have a non-zero reference count even when they are not reachable from a root, so some space
may never be reclaimed. Since web applications are often run for many years, any system that causes
even a relatively minor memory leak of a few kilobytes per minute is unacceptable over such large time
scales.
3.4.3.2  Mark-and-sweep

A mark-and-sweep collector traverses all references and transitive references from the roots and sets a flag in each object reached to indicate that it is accessible and should not be collected. Once all paths have been traversed, any object which is not marked as accessible is deleted. This approach will reclaim all space taken up by inaccessible objects, but requires an expensive traversal of the entire object graph. To make matters worse, modification of the object graph can create newly-inaccessible objects or create new objects that are accessible but not marked, so either the object graph must be made immutable during the marking phase or the algorithm adapted to cater for these occurrences.

3.4.3.3  Generational Garbage Collection

Many garbage collectors for Java and .NET use a copying strategy as opposed to a marking strategy in order to aid compaction of used space. Rather than setting a flag in a reached object, the object itself is moved to a separate area of memory so that there are large, contiguous chunks of free space available after all the unreachable objects have been reclaimed. A generational garbage collector works on the heuristic that newly created objects are more likely to be unreachable than ones that have been around for a long time, so the heap is partitioned into regions corresponding to different lifetimes and objects are copied from region-to-region based on how many collection cycles they have survived. The reachability algorithm would then concentrate on the regions of the heap with shorter lifetimes rather than covering the entire object graph on each invocation of the garbage collector.

In general, database management systems do not give enough low-level access to their data storage to make this technique useful. Often compaction of data is performed on a per-table basis triggered by a single command, such as PostgreSQL’s VACUUM, so combining such an operation with a garbage collection algorithm would be database-dependent. The advantages of separating objects based on their lifetimes is also less beneficial in a persistent store, since the spread of object lifetimes is likely to be much more varied than in heap memory.

Overall, object graphs in persistent stores are potentially much larger than those in memory and exhibit different characteristics to those in a running program. Therefore, the kind of garbage collection used in modern programming languages may not be applicable to the persistence domain. Determining the best algorithm to use in this case is beyond the scope of this project, but any good algorithm should ensure that all unreachable objects will be eventually reclaimed to avoid a disk space leak. For this project, reference counting would provide a fast heuristic for deleting object data on-the-fly, and a full mark-and-sweep collection could be scheduled to be run occasionally in a maintenance task in order to reclaim objects in cycles. Alternatively, Bacon and Rajan suggest a method of combining reference counting and tracing to minimise the number of objects traversed when reclaiming cyclic data in [Bacon and Rajan 01] which could be used.

3.4.3.4  Finalisers, Weak, Soft And Phantom References

Many garbage collection algorithms in Java are complicated by finalisers and weak, soft and phantom references. Finalisers are methods that are automatically run by the garbage collector once an object becomes unreachable but before its data are reclaimed. A problem arises when the finalisation code creates a new reference, termed a phantom reference, from a reachable object to an unreachable one, such that the marking phase must be re-run to determine whether the finalised objects can be reclaimed. Usually, finalisers are discouraged except for freeing native resources, so they will not be considered further for persisted objects.

Weak and soft references are used to provide hints to the garbage collector about when it is acceptable to reclaim accessible objects. In general, they are used in caching to take advantage of free space when it is available, but give way to other objects when free space is tight. Unfortunately, it is difficult to maintain the semantics of weak and soft references for a persistent system, since hard references to objects may exist in the database but not in memory, and so will not be noticed by the Java garbage collector.
Balancing simplicity and efficiency in Web applications

collector. This means that objects referenced weakly or softly may be reclaimed even though they are reachable by strong references. Therefore, for the purposes of this project, weak and soft references will not be considered by the persistence system.

3.4.4 Inheritance

Java allows each class to inherit from only one concrete or abstract class, but it allows it to implement multiple interfaces. Inheritance poses a problem for persistence systems because it allows the dynamic (runtime) type of an object to differ from the static type of its referers. Typically, when fetching objects from a persistent store, it is more efficient to construct a single query to fetch all the object data in one ‘pass’ (i.e. a single communication with the database). However, the table-per-class (TPC) and table-per-path (TPP) techniques for persisting class hierarchies discussed in Section 2.1.5.2 require the table containing the data to be known when the query is constructed. Often, this can be derived from the class name of the object being fetched, but this may not be known until some of the object data has already been fetched. In general, there are three cases to consider:

3.4.4.1 The static type and dynamic type are the same

The static type of a field can be used to lookup the table containing the required object data. Once the object data has been retrieved, the dynamic type of the object will be known. If this is the same as the static type, then all necessary data has been retrieved and the object can be reconstructed.

3.4.4.2 The static type is not an interface and the dynamic type is a subclass of it

The static type is used to lookup the table containing the required object data as before, but this time the dynamic type of the object is found to be a subclass of the static type, so additional fields may exist in other tables corresponding to the subclasses of the static type in the branch of the dynamic type. Therefore, once the dynamic type is known, data from these tables must also be fetched before the object can be reconstructed.

3.4.4.3 The static type is an interface which the dynamic type (or a supertype thereof) implements

Since interfaces contain only static, final fields, which should not be persisted for the reasons stated in Section 3.4.2, a field with an interface as its static type gives no information about the data structure of the objects it may hold. It is also difficult to find the table that stores the object, since many different objects may implement the interface and each may have objects persisted in different tables. A simple solution is to use two passes to find the data; the first to look up the dynamic class of the field in a field lookup table and the other to fetch the necessary data for that class.

A more complicated solution which may perform better in some situations is to examine the class hierarchy to determine which classes implement the interface. If only a single class implements the interface then its table can be found and the data retrieved in a single pass. If multiple classes implement the interface then their tables could either all be queried and their results unioned, which could potentially query many tables unnecessarily, or the least upper-bound of the classes could be found and its corresponding table queried, which would retrieve any data common to all the classes but may require a second pass once the true dynamic type is known.
3.4.5 Generics

The previous section discussed how static type information can reduce the number of database communications required to retrieve an object’s data. Fortunately, since Java 1.5, much more static type information can be included in an application’s source code in the form of generics, which can also be of benefit. The most obvious candidates for this are the classes included in the Java Collections library, since the generic type would allow the collection data and element data to be joined rather than requiring an initial lookup to find the tables corresponding to the types of the elements.

A more general application of this technique is to keep track of the parameters to field declarations and propagate them into the class to determine the most specific static type of its fields. In the example in Figure 3-19, this would reduce the two-pass retrieval needed to reconstruct B objects to a single JOIN between the B, A and Integer tables.

```
class A<T> {  
  T field; 
}
class B {  
  A<Integer> a = 5; 
}
```

**Figure 3-19** - Example demonstrating the use of parameterized types to improve query efficiency.

3.4.6 Class Loading

Java has the ability to load classes at runtime so that new functionality can be added to an application without the need for it to be restarted. This is often used in web application servers to allow web applications to be upgraded with minimal disruption to external users. However, it is not possible to simply replace existing classes with new ones; instead only one version of a class may be loaded per class loader, and instances of a class loaded by one class loader are not assignment compatible with instances of the same class loaded by a different class loader, as shown in Figure 3-20.

```
URL url = new File("C:/temp/*").toURL();
URLClassLoader cl1 = new URLClassLoader(new URL[] { url });
Class c1 = cl1.loadClass("A");
URLClassLoader cl2 = new URLClassLoader(new URL[] { url });
Class c2 = cl2.loadClass("A");
System.out.println(c1.isAssignableFrom(c2));  // Prints "false"
```

**Figure 3-20** - Example code for demonstrating incompatibility between classes loaded by different class loaders.

For a persistence system, it may therefore be not enough to store simply the fully-qualified class name, since this may not uniquely identify the type of object stored. However, in most practical cases we can assume that the class name is unique and only consider multiple classes with the same name when such classes have been upgraded to a later version. This can cause the following problems:

3.4.6.1 A persisted class is upgraded such that the data structure does not change

In this case, the persistence system merely must be capable of producing classes from different class loaders. The new class loader could be passed to the persistence system when an instance is requested in order to ensure that an instance of the upgraded class is reconstructed from the persisted data.
3.4.6.2 A persisted class is upgraded such that its data structure changes

In this case, in addition to ensuring that a correct instance is returned, the schema of the persisted data must be altered to be compatible with the new class structure. Often, this would require an upgrade method to be written by the user that would produce instances of the upgraded class given fields from the old class’ data structure. However, if such a method has not been provided, then any new or altered fields could be added with default values (typically null for references and zero for numeric types) and any removed fields could have their persisted data deleted.

Further problems arise if the persistence layer gets requests to return the same object as an instance of the old class, then later the upgraded class, since one of the properties from Section 3.4.1 states that the exact same object must be returned in both cases. To avoid this problem, it may be necessary to ensure that all references to instances of the old class are lost before the upgrade. In a web application, such instances would typically reside in the session and any database cache, so these must either be flushed or all affected instances converted to use the upgraded classes.

3.5 Summary

This section described some of the techniques used to gather requirements for the specification of this project. The following areas were covered:

- A case study of an enterprise-standard web application was performed, in which various shortcomings of J2EE were exposed and their resulting performance decrease measured. Techniques for avoiding these problems were discussed, which are used as the basis for the performance optimisation requirements stated in Section 4.1.

- The results of a questionnaire were analysed to determine which features are considered important by current web application developers. The questions touched upon a wide range of aspects from the persistence and presentation layers of the project. Their results were used to help form the requirements in Sections 4.1 and 4.2 respectively.

- A prototype implementation was produced and used to help scope the requirements of this project. A number of unforeseen problems with the system design were highlighted and used to ensure that realistic targets were set for each layer.

- The Java language was examined for features that would affect the design and implementation of the persistence system. This produced a set of concrete requirements that a system conforming to the semantics of the language must adhere to. These were used to form the contents of Section 4.1.5 of the project specification.
4 Specification

The goal of this project is to produce a web application framework that improves upon the features and research outlined in section 2 of this report in two areas; firstly, by reducing the burden on the web application developer and secondly by surpassing the performance of existing frameworks. This section breaks down this goal into a number of quantifiable sub-goals that must be achieved by the implementation.

The requirements for this project have been broken down into three distinct layers based on the roles that they perform.

- **The persistence layer** should allow objects to be efficiently persisted to and retrieved from a secondary store with minimal changes to the application code. This forms the main part of this project by addressing concerns that have a large influence on both the efficiency and simplicity of a web application.

- **The presentation layer** should provide an interface for developers to construct web pages that access the features of the persistence layer and display the persisted data to the user in the form of a web page.

- **The server** should intercept web requests, manage the loading of classes necessary to process them, perform web-server-related error handling and return information to the user.

The interactions between each of the above components and the web application itself are best described by their roles in the processing of web requests. The server should receive requests and lookup the classes necessary to service them, termed *page classes*, from the web application. The page classes will invoke methods on other classes in the web application, which may in turn store, retrieve or update objects in the persistence layer. To produce output, the page classes should construct instances of classes from the presentation layer and return them to the server. The server should then request data from these classes in the necessary format and return it to the user.

![Figure 4-1 Overview of the interactions between components of the framework and the web application](image-url)
4.1 The Persistence Layer

The high-level requirements of the persistence layer are to provide a store in which:

- objects may be placed and a unique ‘handle’ for that object obtained
- objects may be retrieved given their handle
- objects placed in the store and dominated\(^{20}\) by the persistence system are no longer required to be stored in main memory

This project also requires the additional ‘soft goal’ that:

- the average time between an object store or retrieve request and that request being fulfilled is as small as possible

The requirements elicitation and background sections described how these high-level statements were refined by constraining the types of objects to be considered to those in the Java language and the form of storage to a relational database. This section further refines these goals into a set of minimum requirements for providing an orthogonal persistence system for the Java language.

4.1.1 Overview

This persistent system should be different from existing systems discussed in the background section in that:

- It should be capable of persisting objects regardless of their type. This adds the following constraints to the persistence system:
  - Persistable class files should not have to be run through a manual enhancement stage.
  - Persistable classes should not have to be annotated with additional information for the purpose of guiding the persistence system.
  - Persistable classes should not have to be declared to the persistence system prior to the first instance of that class being persisted.
- It should run on the unmodified Sun Java 5 Virtual Machine.
- It should use the unmodified Java 5 Language and Java 5 Compiler.

Modifications to the Java Language, Compiler and Virtual Machine are unacceptable since they would discourage widespread use of the project and make it incompatible with future versions of Java.

---

\(^{20}\) An object \(o_1\) is said to be \(dominated\) by another object \(o_2\) if all paths from the root object to \(o_1\) go through \(o_2\). In this case, the persistence system dominates an object if it holds the only strong reference to that object.
4.1.2 Persistence Interface

The persistence layer should provide a suitable interface for fetching and retrieving Java objects. In addition to this, it should take advantage of generic type information to optimise the storage and access of these objects where necessary. **Figure 4-2** shows the interface that the persistence layer should provide to the other layers of the project to provide these services.

```java
interface PersistenceLayer {
  Handle storeObject(Object object, Type type);
  Object retrieveObject(Handle handle, Type type);
}
```

**Figure 4-2 - Expected interface for the persistence layer.**

4.1.3 Observational Equivalence

I define the notion of observational equivalence between objects in order to describe acceptable implementations of orthogonal persistence. Two objects are said to be observationally equivalent if there is no means to distinguish between them within the Java language. However, for the purposes of implementation this is relaxed to exclude comparisons involving `System.identityHashCode(..)` and heap space analysis using the Java Virtual Machine Tool or Debug Interface.

4.1.4 Persistable Types

The aim of an orthogonal persistence system is to allow instances of all data types to be persisted and retrieved such that the retrieved instances are observationally equivalent to the persisted instances. Unfortunately, there are many types in Java that access native data or declare fields as transient which prevent a general algorithm to handle all data types from being constructed. Therefore, I have constrained the minimum set of supported types as follows, with further explanation later in this section:

- All Java classes that have a no-argument constructor and do not use the `static`, `final` or `transient` modifiers for any of their fields.
- All primitive Java types

  `int, float, short, double, byte, boolean, char` and `long`

---

21 `System.identityHashCode(..)` returns a unique number for each object, typically computed based on the internal address of the object within the JVM. This is not supported because such a number could be stored between JVM executions, for which the same object is not guaranteed to be at the same address.

22 Using JVMTI or JVMDI it is possible to access and store information about objects allocated in the heap between JVM executions. This is not supported for similar reasons to `System.identityHashCode(..)`. 

---

75
• All autoboxed primitive Java types
  
  Integer, Float, Short, Double, Byte, Boolean, Character and Long

• An implementation of the following standard Java 5 interfaces:
  
  List, Set, Map, Image

• Array types where the component type is a persistable type

Classes that have no-argument constructors but use the static, final or transient modifiers for some fields should be persisted by omitting those fields; however their retrieved instances are not guaranteed to be observationally equivalent to their persisted instances.

4.1.4.1 Persistence Of Java Class Instances

Instances of Java classes should be persisted by recursively persisting each of their fields and the fields of its direct superclass and any transitive superclasses. Fields marked with the static, final or transient modifier should be omitted (see Section 3.4.2).

Instances of Java classes should be reconstructed by reconstructing each of their fields and the fields of its direct superclass and any transitive superclasses. Fields marked with the static, final or transient modifier should not have their values reconstructed and should be left unmodified.

Field Handling Techniques

The persistence layer should support the persistence of fields using the Foreign Key Aggregation and Single Table Aggregation techniques (see Section 2.1.5.1):

• Field values persisted using Foreign Key Aggregation are stored in tables corresponding to the runtime types of the values. The field values and classes in which the field is declared are related by a foreign key reference across their corresponding table schemas.

• Field values persisted using Single Table Aggregation are stored in the table corresponding to the class in which the field is declared.

Assignment of these techniques should be done at the field level, allowing fields within the same class to use a different persistence technique.

The persistence layer should be able to alter the technique used for persisting a field. Any existing persisted values for such a field should be redistributed to use the newly-assigned technique. This process should be performed as part of the maintenance task (see Section 4.1.7).
Inheritance Handling Techniques

The persistence layer should support the persistence of classes related through an inheritance hierarchy using the Table Per Class, Table Per Inheritance Path and Table Per Inheritance Hierarchy techniques (see Section 2.1.5.2):

- Class instances persisted using Table Per Class have their persistable field values stored in tables corresponding to the classes that the fields were declared in. All data corresponding to the same instance is related by a foreign key reference between the table schemas corresponding to the instance’s superclasses.

- Class instances persisted using Table Per Inheritance Path have all their persistable field values stored in the table corresponding to the instance’s runtime type.

- Class instances persisted using Table Per Inheritance Hierarchy have all their persistable field values stored in the same table as instances of the superclasses of its runtime type.

Assignment of these techniques should be done at the class level, allowing instances of different classes from different parts of the inheritance hierarchy to use different techniques.

The persistence layer should be able to alter the technique used for persisting a class. Any existing persisted instances of such a class should be redistributed to use the newly-assigned technique. This process should be performed as part of the maintenance task (see Section 4.1.7).

Use Of Static Type Information

When reconstructing an object, the runtime type will not be known until at least partial object data has been retrieved from the database. The persistence layer should minimise the number of SELECT statements that must be executed to reconstruct an object by using any static type information provided:

Fields should be retrieved using a single SELECT statement by adding a JOIN between the table containing the field reference and the table containing the field value (if Foreign Key Aggregation was used).

If an object and all of its field values have the same runtime type as their static type then only a single SELECT (with multiple JOINs) should be necessary to reconstruct that object.

If an object’s runtime type is different from its known static type then a second SELECT may be constructed to fetch the fields of classes between the static type and the runtime type in the class hierarchy.

Given the above rules, the number of SELECTs required to reconstruct an object graph containing N objects should be constrained by:

- a minimum of 1, which occurs when the runtime type of each object is the same as the known static type of that object.

- a maximum of N+1, which occurs when the runtime type of each object is different from the known static type of that object.
Cycles In The Static Object Graph

The persistence layer should be able to persist instances of classes that directly or indirectly reference themselves. Fields that are part of these cycles may be omitted from the Single Table Aggregation technique in order to avoid creating infinitely large tables.

Cycles In The Dynamic Object Graph

The persistence layer should be able to persist objects that refer to themselves in one or more of their fields, or transitively through the fields of other objects that they refer to. These objects may be reconstructed by multiple SELECT statements to avoid creating statements containing an infinite number of JOINs.

4.1.4.2 Persistence Of Primitives

The persistence layer should take advantage of the immutability of primitive types to reduce the number of operations required to retrieve and update them. Such reduction techniques should include:

- Automatic inlining of primitive values where they are referenced from persisted objects.
- No generation and storage of a unique handle for primitive values that are referenced from persisted objects.
- Updates of a primitive field value need only alter that field’s value in the database and not any other fields which refer to the same value. This is because immutable values are only ever replaced, which does not affect the other references.

4.1.4.3 Persistence Of Autoboxed Primitives

Autoboxed primitives should be handled identically to standard primitives. The rules for observational equivalence are relaxed for autoboxed primitives such that pointer (double-equals) equivalence need not hold. This is acceptable because the Java Language Specification makes no guarantees on the pointer equivalence of autoboxed types.

4.1.4.4 Persistence of Lists

The commonly-used List implementations provided in the Java 5 Collections Framework all store their element data in transient fields that will not be persisted by the general class instance algorithm described in this section. Each implementation also has different performance characteristics that make a general List-handling algorithm less desirable, since the method in which it traverses the List may impact its efficiency.
To demonstrate support for List instances, the persistence layer should provide an efficient means of persisting one of the List implementations in the Java 5 Collections Framework. The persistence strategy used should ensure that the following properties hold:

- A List instance is persisted by recursively persisting each of its elements.
- The order of elements within a List is preserved by the persistence mechanism; however the actual ordering of the rows in the database may differ from this to allow for efficient insertions.
- Elements of the List should be fetched lazily to prevent retrieving the entire contents of the List when it is requested.

### 4.1.4.5 Persistence of Sets

Standard Set implementations suffer from the same problems as Lists except that they prohibit multiple references to the same element and some do not require the order of their elements to be preserved. Not having to maintain element ordering may significantly increase the performance of fetching a Set by eliminating ORDER BY operations and should therefore be used where possible. Since some implementations require order to be preserved a general algorithm for all implementations is undesirable.

To demonstrate the additional efficiency provided by ignoring element ordering, the persistence layer should provide an efficient means of persisting one of the unordered Set implementations in the Java Collections Framework. Lazy loading of Set elements is not necessary since the checks needed to ensure that multiple references to the same element are disallowed will require all elements to be present, limiting the effectiveness of the technique.

### 4.1.4.6 Persistence of Maps

Standard Map implementations suffer from the same problems as described for Sets, except that each Map entry must store a reference to both its key and value objects. To demonstrate support for Map instances, the persistence layer should provide an efficient means of persisting one of the Map implementations in the Java 5 Collections Framework.

### 4.1.4.7 Persistence of Arrays

Arrays should be persisted by recursively persisting each of their elements, ensuring that the ordering of the elements is preserved. Lazy loading for arrays is not a requirement since detecting array accesses in Java is difficult within the constraints of using the unmodified Sun Java Virtual Machine.

### 4.1.5 Maintaining Java Semantics

The persistence layer should maintain Java semantics for objects while they are in the database such that persistent instances behave identically to non-persistent instances. Section 3.4 described a number of features of Java and how they apply to persisted objects; this section expands on those by listing the minimum requirements for a persistence system to implement these features in the persistence domain.
4.1.5.1 Types

After an instance of a type has been stored in the database, requesting that object should produce an instance of the same type.

The persistence system should be robust to changes made to a type between JVM executions. The following properties should hold:

- Altering a class to add, remove or alter methods should have no effect on the persisted instances of that class.
- Altering a class to add new fields should add default values for those fields to persisted instance of that class.
- Altering a class to remove fields should remove the values of those fields from persisted instances of that class when the maintenance task is next run.
- Altering a class to add a superclass to its inheritance hierarchy should add default values for the fields of the superclass to persistent instances of the class.
- Altering a class to remove a superclass from its inheritance hierarchy should remove the values of the fields of that superclass from persisted instances of the class when the maintenance task is next run.
- Altering the superinterfaces of a class should have no effect on the persisted instances of that class.

It is sufficient to store instances of a class based on the fully-qualified name (package name and class name) of the class. The persistence system is not required to distinguish between classes with the same fully-qualified name loaded by different class loaders.

4.1.5.2 Reference Semantics

Storing an object and then fetching it produces an object that is observationally equivalent to the original. If both objects exist within the same JVM then they must be ‘double-equals’ equivalent (such that \( \text{object1} == \text{object2} \) is true). This may be relaxed for immutable objects to allow for storage and update optimisations.

Updating the in-memory data of an object updates the persisted data. At the point of JVM termination, persisted objects should contain the same data as their in-memory counterparts (if they exist).

The semantics of soft and weak references cannot be successfully preserved for the reasons discussed in Section 3.4.3.4, so may be omitted by the persistence system.

4.1.5.3 Garbage Collection

Persisted objects should remain available until no longer reachable by traversing paths from persisted fields.

Unreachable object data in the database should be reclaimed at regular intervals. Reference counting should be used to reclaim objects ‘on-the-fly’ where possible.
All unreachable object data must eventually be reclaimed. This should be achieved by occasionally performing a complete sweep of the object graph using a tracing collector, or by sweeping only the objects that have not been reclaimed by reference counting due to cycles using a hybrid collector.

### 4.1.6 The Heuristic Optimiser

As already stated in section 4.1.4.1, the persistence layer should be capable of persisting references using both Foreign Key Aggregation and Single Table Aggregation and inheritance hierarchies using Table Per Class, Table Per Inheritance Path and Table Per Inheritance Hierarchy techniques. It must also be possible to apply these techniques to different parts of the class graph, rather than applying the same technique to all classes and fields. A heuristic optimiser should also be provided that will attempt to find the mapping of techniques to each part of the class graph that provides the most efficient overall access and update times for instances of those classes.

The overall access and update times for instances of a class should be based on the average `SELECT`, `INSERT` and `UPDATE` times for instances of that class and biased by the number of times instances are accessed and updated respectively, i.e.:

\[
T_C = S_C \times N_{SC} + I_C \times N_{IC} + U_C \times N_{UC}
\]

where
- \(T_C\) = Overall time for class \(C\)
- \(S_C\) = Average `SELECT` time for \(C\) instances
- \(N_{SC}\) = Number of times instances of \(C\) have been accessed
- \(I_C\) = Average `INSERT` time for \(C\) instances
- \(N_{IC}\) = Number of times instances of \(C\) have been inserted
- \(U_C\) = Average `UPDATE` time for \(C\) instances
- \(N_{UC}\) = Number of times instances of \(C\) have been updated

**Figure 4-3** - Expected cost function for evaluating the overall access and update times for instances of a class \(C\).

A set of suitable heuristics should be included to estimate the values of \(S_C\), \(I_C\) and \(U_C\) for the different combinations of persistence techniques used for a class and suggest a combination of techniques that minimises \(T_C\). These heuristics should take into account the following properties:

- Single Table Aggregation improves `SELECT` performance for the referring instance but causes field value duplication such that multiple `UPDATE`s must be performed when an instance is altered.
- Foreign Key Aggregation reduces field value duplication such that only a single `UPDATE` must be performed per instance, but each `SELECT` statement must contain multiple `JOIN`s across the tables containing the required field values.
- Table Per Inheritance Path reduces the duplication of class data such that only a single `UPDATE` need be performed when an instance is altered, but may require multiple `JOIN`s when accessing the data.
- Table Per Inheritance Hierarchy may produce large tables with many `NULL` values which could increase time of both `SELECT` and `UPDATE` operations upon that table.
- Table Per Class reduces duplication of class data, decreasing the number of `UPDATE`s that must be performed when an instance is altered, but increases the number of `JOIN`s required between the tables corresponding to each class in the required branch of the class hierarchy.
If the techniques used for persisting a class are altered, the corresponding table schema for that class must be updated to match the mapping specified by the new techniques. Any instances that have already been persisted to that table must be stored using the new mapping, which may be a very time-consuming operation for large numbers of instances. Therefore, the persistence layer should perform all necessary schema changes together as part of the maintenance task. It may also be desirable to add a ‘frictional’ component to the heuristics that bias against changing a table schema if the table contains a large number of instances.

4.1.7 Maintenance Task

The maintenance task should be responsible for running occasional, heavyweight operations on the database at times when database load is low. It should perform the following tasks:

- If an alternative technique to persist a field of a class or part of the inheritance hierarchy is suggested by the heuristic optimiser then the maintenance task should apply this change by redistributing field and class data according to the new mapping.
- Any unreachable persistent instances that have not been deleted by reference counting should be deleted by a tracing collection algorithm.
- If fields have been removed from a class in the Java domain then any corresponding values for those fields should be removed from instances of that class on the database.

It is sufficient for the maintenance task to prevent access to the database while it runs in order to avoid concurrent queries returning inconsistent results. A better solution which intercepts queries made during schema changes and directs them to the necessary tables could be investigated if time allows.

4.2 The Presentation Layer

The high-level requirements of the presentation layer are to provide an interface to be used by web application developers to:

- store and retrieve Java objects between web requests.
- construct a web page based on dynamic data and with a clear separation between content and style.

This project also requires the additional ‘soft goal’ that:

- the interface presented to the developer be simple and intuitive to use.

The questionnaire results discussed in the requirements elicitation section discussed the importance of additional goals and refinements of these goals (see Section 3). This section uses that information to give a set of minimum requirements that must be met by the implementation of this project in order to provide an acceptable presentation layer.
4.2.1 Web Page Interface

The presentation layer is responsible for allowing developers to easily write dynamic web applications that can be plugged into the implementation of this project. The minimum requirements for achieving this task are:

- providing the developer with an interface for denoting a particular class as a web page class and associating the class with a particular URL.

- registering the web page class and the associated URL with the server layer, such that the class is notified of visits to that URL.

- forwarding the document returned by the web page method to the server layer so that it can be displayed to the user.

4.2.2 Persistent Data Interface

Most objects in web applications can be grouped into three categories depending on their lifetimes (the time between creation and when they become unreachable):

- Request data are only available during the lifetime of the request being serviced. This includes user data submitted via GET or POST operations\(^2\) from a web page form and temporary objects constructed while the request is being serviced. Once the request is complete, all request data are made unreachable.

- Session data are partitioned into sections based on each current, unique user of the web application. This typically includes user preferences and login information. A user’s session is invalidated when the time since their last request to the web application is greater than the session timeout value. When this occurs, all session data are made unreachable.

- Database data contains the objects that are shared between all users of the site (barring access controls explicitly imposed by the web application) and exist until made unreachable by the web application code. Database data are typically tailored to the particular web application, but may include the list of authorised users, message board posts and gallery images.

The presentation layer must provide the developer with a consistent interface for storing objects using each of the above strategies. This should be implemented by providing a set of field annotations that determine the lifetimes of objects stored within those fields. Three different annotations should be made available:

- The @Request annotation should denote a field that holds GET or POST data for the current request. Before the web page method is invoked, such fields are filled with the GET or POST data which has the same key as the field name. If both GET and POST data exists for the same key then the GET data value should be used. If neither GET nor POST data exists for that key then the field should be set to null. Since request data only lasts the duration of the

\(^2\) GET and POST data is represented as a mapping between keys and values, both of String type, that are passed by the user with the web request. GET data is typically passed as a sequence of ‘name=value’ pairs in the URL delimited by ampersands. POST data is passed as part of the request header and is typically provided by form submission, where each of the keys corresponds to a particular input field in the form.
request, all fields marked with the @Request annotation should be nulled after the request has been serviced.

- The @Session annotation should denote a field that holds the contents of a variable from the user’s session. Before the web page method is invoked, such fields should be filled with the session data that corresponds to the field name, or set to null if such data does not exist. Once the request has been serviced, any values placed in fields marked with the @Session annotation are stored in the user’s session, keyed by the field name.

- The @Database annotation should denote a field that holds a persistent object. The object should be associated with the field’s name and static type, such that those with the same name and static type will share their values. Before the web page method is invoked, the persistent object corresponding to the name and type of each @Database field should be requested from the persistence layer and assigned to the field. If a persistent object matching a field’s name and type does not exist then the field should be set to null. Once the request has been serviced, any values placed in these fields should be passed to the persistence layer to be made persistent (or updated if they are already persistent).

### 4.2.3 Document Interface

The main role of the presentation layer is to provide an interface for the developer to construct dynamically generated web pages. The content of these pages should go through the following stages before it is passed to the server layer to be displayed:

- the web page classes should build up the page content in the form of a document tree.
- a sequence of style transforms should be applied to the document tree to decorate the content with the appropriate style elements that dictate how the content should be displayed.
- a formatter should walk the styled tree and output it in the necessary format to be displayed to the user.

To achieve this, the presentation layer should provide a set of interfaces and example implementations for the document tree, the style transforms and the formatters. The following subsections describe the individual requirements of each.

#### 4.2.3.1 Document Tree

Pages constructed using the document tree may also need to be viewed in a variety of different formats, such as HTML, RSS and PDF. Therefore, the document tree must be able to represent content in each of those formats. To achieve this, a tree structure based on the XML model should be used, whereby:

- each node contains a (possibly empty) sequence of ordered child nodes.
- each node contains a (possibly empty) set of attribute data in the form of name-value pairs.

For the purposes of this project, a set of node implementations should be provided for representing a core set of HTML tags along with their associated attributes. The tags supported should include:

\[
\text{html, head, title, body, table, tr, td, a, form, input, textarea, div, img, h1, h2, ul, li, p, span}
\]
When constructing a dynamic web page, many components of the page will be common amongst several pages of that site. Therefore, it should be possible for the developer to create custom nodes that represent this common functionality. These nodes should then be expanded into low-level form by the style transforms (see Section 4.2.3.2).

An example set of high-level node implementations should be provided to represent the following:

- tabular data, where the cell data and column count are stored and are expanded to table, tr and td tags by a corresponding style.
- data submission forms, which store information about the required fields and data types and are expanded to form, input and textarea tags with Javascript validation.
- paginated lists, which take a list of objects and displays a sub-list that the user can navigate through.

4.2.3.2 Style Transforms

One of the requirements of the document tree is that it may contain high-level custom nodes which must be expanded to low-level, format-specific nodes before the page data is passed to the user. The logic for expanding high-level nodes is encapsulated in the style transforms. It is up to the presentation layer to provide a simply interface for developers to write their own style transforms for expanding their custom nodes.

Style transforms should be capable of the following:

- searching the tree for nodes of particular types.
- once such a node is found, replacing it with a different node.
- accessing and modifying any other part of the document tree.

The last point is necessary for providing developers with the freedom to tailor the structure of their documents without being constrained by the low-level format it will be converted to. For instance, a paginated list may wish to modify the head tag of an HTML document to add Javascript handler for when the user clicks to access the next or previous list pages.

Style transforms do not necessarily convert high-level nodes directly into low-level ones. They may produce intermediate high-level nodes for other transforms to convert, allowing high-level components of the page to be grouped together and reused as well as low-level ones. Therefore, each page must be processed by a sequence of style transforms before all of its nodes are low-level. It must also be possible for multiple style transforms to exist for converting the same node to different formats. For example, one style may convert a paginated list to a set of HTML tables, while the other converts it to plaintext for use in an RSS feed.

4.2.3.3 Formatters

Formatters should traverse a document tree and write it out to a byte stream in the format that the user should receive. For XML formats such as HTML and RSS, a single formatter should be provided for writing out the tree in XML form. The presentation layer should also provide the ability to plug in custom formatters for non-XML formats such as PDF. This requires that all formatters conform to a standard, public interface specified by the presentation layer.
4.2.3.4 Style Managers

A style manager is responsible for taking a document tree and producing a byte stream containing the data in a specific format to be passed to the user. It specifies the sequence of styles to be applied to the document tree, the formatter to be used to convert the tree into a byte stream and the MIME type of the resulting data. The presentation layer should provide a standard interface for developers to write custom style managers for, as well as an example implementation for converting documents into an HTML format.

4.3 The Server Layer

The server layer is responsible for handling all interaction with the user and managing the configuration of web applications. The high-level goals of the server layer are to:

- dynamically compile and load web applications.
- intercept HTTP requests and forward them to the correct web application.
- provide a user interface for configuring the application.

The remaining subsections discuss the individual requirements for fulfilling each of these tasks.

4.3.1 Loading of Web Applications

Many web developers are used to using scripting languages that reflect alterations made to the source code immediately in the running application. The server layer should offer similar functionality by making use of Java’s dynamic linking capabilities. The following features should be supported:

- developers should be able to specify a development directory containing the source code or compiled class files for their web application.
- modifications to files in this directory while the server is running should be automatically detected by the server layer.
- if Java source files have been modified, they should be compiled and the resulting class files stored in a temporary directory.
- upon modification, the latest versions of the class files should be loaded and instances of the old versions discarded.

It should be noted that instances of new versions of a class are incompatible with instances of old versions (see Section 3.4.6). It is therefore acceptable for all active sessions to be invalidated when modified classes are loaded in order to ensure that instances of old and new versions of a class do not co-exist.

---

24 Multipurpose Internet Mail Extensions (MIME) content types are used by a web browser to determine how a web page should be rendered. It is therefore important that the MIME type returned by the presentation layer correctly matches the format of the document.
4.3.2 Intercepting HTTP Requests

The server should intercept HTTP requests and invoke the relevant page classes from the presentation layer for servicing those requests. It is responsible for determining:

- which web application should handle the request.
- which page class of the web application should receive the request.
- which style manager should be used to format the response.

To achieve this, the server layer must keep a set of rules that map requests to web applications, page classes and responses. These rules should be in the form of URL patterns that can be matched against the URL of each request.

![URL Components](http://localhost/dir1/dir2/File.html)

**Figure 4-4 - The different components of a URL.**

Although custom URL mapping rules should be supported, the default rules for mapping URLs to web applications, page classes and styles should be as follows:

- For web applications, the URL pattern should correspond to a context path; a sequence of subdirectories from the host name under which the pages of that web application are accessible.
- For page classes, the URL pattern should match the file name against the name of the class.
- For style managers, the URL pattern should match against a file extension (for example, a pdf extension might map to style manager for producing PDF documents).

4.3.3 User Interface

The server layer should provide a user interface that provides the following features:

- allows web applications to be added, removed and configured.
- allows database connections to be added, removed and configured.
- allows logging messages to be viewed.
- allows profiling information to be viewed.
- allows the listening port, on which the server receives HTTP requests, to be set.
Using this interface, the developer should be able to customise the following aspects of each web application:

- the context path at which the pages or resources of the web application can be accessed.
- the development directory, in which the page class or source files reside.
- the resource directory, in which additional resources (images, static pages) reside.
- the database connection used to store persistent data for this web application.

The interface should also provide methods for altering the following settings for each database connection:

- the host name or IP address of the database server.
- the name of the database to be used.
- the username and password to be used to login to the database.

A logging interface should be provided which each web application may use to store error or information messages to be displayed to the user. A log viewer should be provided for viewing messages stored using this service.

A simple profiling interface should be provided for monitoring the time taken to fulfil web requests for each page class of each web application. These results should either be provided in a form that can easily be graphed by an external application, or graphs of the information should be constructed and displayed by the server layer itself.
5 The Persistence Layer

A focus of this project is to provide an orthogonal persistence mechanism for Java in order to simplify the storage of persistent data for web applications. This mechanism also uses a number of strategies to increase the efficiency of access to persisted objects by trying to find the best object-relational mapping for each class based on how its instances are accessed and updated. The task of finding this mapping is usually left to experienced database maintainers who manually specify the SQL needed to persist and retrieve objects. In this project, this is managed by a complex feedback system that monitors object usage and adjusts the mapping accordingly based on a set of heuristics.

Finding a general algorithm to replace the manual task of finding an efficient object-relational mapping is complicated by the large number of competing concerns involved. An example of this is the problem caused by the assignment compatibility between subclasses and superclasses in Java. The data requirements of an object are dictated by the fields declared in that object’s runtime type, so it seems reasonable to persist such an object to a table specific to that type. However, that object may well be referenced from a field of its supertype, in which case the type of the field no longer provides an exact table in which the field value may be found, but a set of tables corresponding to its subtypes. To make matters worse, some field values may be inlined into the tables of the referring class, providing an even larger set of tables to be searched.

This section highlights a number of similar problems found during the progress of this project and discusses how they were solved. Each subsection begins by covering the initial design for the storage and retrieval of objects, then describes the limitations found and justifies the evolution of the design over the course of the project. Finally, the implementation of the persistence layer is described along with limitations found with the tools and interfaces used.
5.1 Overview

Some languages such as SmallTalk express all their data types in terms of a single data model (the 'class'), which allows all data to be inspected via the same interface. Unfortunately, Java's data types are divided into data type groups such as primitives, arrays and classes, each of which must be accessed using a different set of methods from the Java Reflection API. This impacts the design of the persistence layer since there must be dedicated code for accessing the data from each of these groups.

However, as described in Section 4, dedicated code for particular types must also be used within these groups. This is because many classes in the Java Standard API declare their data as transient to indicate that they should be persisted using a custom persistence mechanism. To cope with this, the overarching design of the persistence layer contains a system of pluggable classes for handling persistence for a particular set of types.

At a basic level, a core SqlProvider class provides operations for constructing SQL to create tables, insert data, update data and access data common to all data types. This is then subclassed to provide a persistence strategy specific to a particular group of data types. These plug in to the MasterSqlProvider class, which is given requests to persist and fetch objects and is responsible for delegating these requests to the correct SqlProvider based on the object's type.

Each specific SqlProvider subclass implements a boolean accepts(Type t) method which returns true if the given type can be persisted by that subclass. This method makes no guarantees about the efficiency or correctness of such persistence, and it is often the case that a single type will be accepted by multiple SqlProviders. The Object[] class, for example, is accepted by both the ObjectSqlProvider and the ArraySqlProvider. The former would attempt to persist Object[] instances like any other class, by recursively persisting their field values, but would neglect to store the element data. In these cases, it is the most specific SqlProvider that should be chosen, so the MasterSqlProvider must maintain an ordering of specificity to avoid such ambiguities.

This relatively simple design of the persistence layer is complicated by the need to cater for reference handling, inheritance, lazy loading and aliasing of objects. The following sections address how these issues impact the storage and retrieval of instances for each of the supported type groups. Methods for choosing which optimisations to apply to which classes are then discussed, and in the final section the overall design is summarised and potential improvements suggested.
5.2 Object Storage

This section concentrates on the design and implementation of the classes concerned with persisting objects to the database. When an object is passed to the persistence layer to be stored, first the database tables necessary for holding the object data must be created if they do not already exist and then the data must be copied to the database domain in the format described by the object-relational mapping. Each of the following subsections describes how these tasks are achieved for each of the required data type groups.

When deciding which object-relational mapping to use, several different options can be chosen. For references, either the Foreign Key Aggregation or Single Table Aggregation technique could be used, while for inheritance hierarchies either of the Table Per Class, Table Per Inheritance Path or Table Per Inheritance Hierarchy strategies could be used. This section discusses how each of these techniques were implemented in the persistence layer to allow a large degree of flexibility in the object-relational mapping.

5.2.1 Storage Of Class Instances

When an instance of a class is first passed to the persistence layer, it is assigned an object ID (oid) value which is unique to all instances stored in that database. If this is also the first instance of that particular class to be persisted, the persistence layer will create a new mapping for that class which defaults to using Foreign Key Aggregation on all fields. For each database, such a mapping is only ever created once per class and then stored in a special table so that it can survive JVM termination. The table schema for the mapping is then created by constructing and executing a `CREATE TABLE` command. For Foreign Key Aggregation, each of the fields, including those of primitive type, will map to a single column in the class’ table, which will contain the oid of the field value stored in that field. For a PostgreSQL database, this column will always be of type `INT8`, which is equivalent to the `Long` type used for storing oids in the Java domain.

Once the class table has been created, the instance can then be inserted into it. This is achieved by constructing an `INSERT` statement containing the oids of each of the field values. To get the oid of a field value, the value must first have been persisted to the database, so the storage method is called recursively. This produces a table layout like the one shown in Figure 5-2, where every distinct object maps to a row in a separate table, linked by their oid values. So far, this algorithm does not address the problem of aliasing, where a single instance may be referenced from multiple places (as shown by the first and third B instances in the example referencing the same A instance). This is catered for by an object cache which maintains a mapping between objects in the Java domain and their corresponding data in the database domain and is discussed in Section 5.2.1.2.

```java
class A {
    int ai;
    public A(int ai) { this.ai = ai; }
}

class B {
    A a;
    public B(A a) { this.a = a; }
}

B b1 = new B(new A(57));
B b2 = new B(new A(90));
B b3 = new B(b1.a);
store(b1); store(b2); store(b3);
```
Balancing simplicity and efficiency in Web applications

Figure 5-2 - Example code and the resulting table layout for persistence of objects using Foreign Key Aggregation.

5.2.1.1 Single Table Aggregation

As an alternative to Foreign Key Aggregation, the persistence layer also supports Single Table Aggregation as outlined in the specification section. Initially, this was achieved through a class-based inlining approach, but this was found to be quite inflexible and was replaced by a path-based inlining approach. Both strategies are mentioned here so that the differences between the two can be highlighted.

Class-based inlining

The specification of reference persistence states that the persistence layer should support different persistence strategies for different fields. In the initial implementation, this was achieved by associating each class with a TypeInfo object that stored metadata concerning the mapping of fields. Each field of the class had a corresponding boolean value in the TypeInfo object to indicate whether the field should be persisted using Foreign Key Aggregation or Single Table Aggregation. For conciseness, fields that use Single Table Aggregation were denoted inlined fields, since their value is inlined into the table of the referring class.

One of the problems with this approach is that the CREATE TABLE and INSERT statements constructed for a particular type now depend on the types of the fields. For inlined fields, all the columns in the table corresponding to the field type must appear in the referring type’s table. To create a table for a particular class, first the SqlProvider for that class constructs a CREATE TABLE command containing only the default oid column. The declared fields of the class are then iterated over and, for each field, the SqlProvider for its declared type is obtained and passed the CREATE TABLE statement. Each of these SqlProviders then modify the statement to add the columns needed to inline their types.

```java
class A {
    int ai;
    public A(int ai) { this.ai = ai; }
}
class B {
    A a;
    public B(A a) { this.a = a; }
}
B b1 = new B(new A(57));
B b2 = new B(new A(90));
B b3 = new B(b1.a);
setInlinedFields(B.class, { B.a });
setInlinedFields(A.class, { A.ai });
setInlinedFields(int.class, { int.value });
store(b1); store(b2); store(b3);```

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>a</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>b</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>57</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>90</td>
<td></td>
</tr>
</tbody>
</table>
Figure 5-3 shows the different parts of the statement contributed by each of the different SqlProviders for creating a class table in which all fields are inlined. Certain precautions must also be taken to ensure that there are no naming conflicts between columns if a class contains two inlined fields of the same type (such as if the B class was modified to contain two A fields). These issues are discussed in more depth in 5.2.1.3.

With the class-based approach, inlining a field is a recursive process such that if a class c1 contains an inlined field of type c2 and class c2 contains an inlined field of type c3 then the c3 values will be inlined into the c1 table. However, this means that the resulting table structure depends on the order in which the inlining occurs, since inlining a field into c2 after it has already been inlined into c1 will produce a different table schema. A further problem occurs with cycles in the static object graph, since inlining a class into itself will produce an infinitely long CREATE TABLE statement.

Aside from these problems, this approach is also restrictive in the amount of control a class has over its transitively inlined fields. For example, if the A class in Figure 5-3 has ai as an inlined field, then denoting a as an inlined field of the B class will automatically cause the ai field to be inlined into the B table. In practice, the ai field may be very rarely accessed in the context of a B object, in which case inlining the ai value into the B table may cause a decrease in performance.
Path-Based Inlining

Whereas class-based inlining provided the option to inline a field and all transitive fields based on the class of that field, path-based inlining allows specific paths of field traversals to be inlined. A path is defined as a sequence of fields, where each field is declared in the type of the previous field in the sequence. The TypeInfo metadata was extended to include a set of paths denoting the data values to be inlined from all immediately and transitively accessible fields.

To create a class table, the SqlProvider for that class constructs a CREATE TABLE statement containing the default oid column as before, but this time it iterates through the set of inlined paths. For each path, the first field in the path is inspected and the SqlProvider for its declared type obtained. The CREATE TABLE statement is then passed to this SqlProvider along with the remainder of the path, so that it can modify the statement to add its own columns. The process then recurses, with that SqlProvider inspecting the next field in the path and passing the statement and remainder of the path to the SqlProvider that corresponds to the field’s type.

A similar strategy is used to construct INSERT statements, but this time each SqlProvider in the chain receives the INSERT statement, the remaining path and the current object to be inserted. It modifies the INSERT statement to add the object’s oid, then for the next field in the path, looks up the value of that field in the current object and passes that value to the next SqlProvider. In this way, as each path is traversed, the corresponding fields of the object are also traversed to obtain the required field values. Figure 5-6 shows an example of this process in action.

```java
class C {
    A a;
    B b;
    public C(A a, B b) { this.a = a; this.b = b; }
}
class D {
    A a;
    C c;
    public D(A a, C c) { this.a = a; this.c = c; }
}
setInlinedPath(D.class, { D.a.ai.value, D.c.a.ai.value });
D d1 = new D(new A(42), new C(new A(29); new B(new A(31)));
store(d1);
```

<table>
<thead>
<tr>
<th>oid</th>
<th>a</th>
<th>a ai</th>
<th>a ai value</th>
<th>c</th>
<th>c a</th>
<th>c ai</th>
<th>c ai value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>42</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>29</td>
</tr>
</tbody>
</table>

**Figure 5-5** - Example code and the resulting table layout for persistence of objects using path-based inlining.
2. The ObjectSqlProvider for the B type adds the oid of the d1.a value to the INSERT statement.

3. The ObjectSqlProvider for the A type adds the oid of the d1.a.ai value to the INSERT statement.

4. The PrimitiveSqlProvider for the int type adds the value of the d1.a.ai field to the INSERT statement.

1. The ObjectSqlProvider for the D type constructs the INSERT statement with the oid of d1. The addInlinedPath(...) method is then invoked for the first inlined path.

5. The ObjectSqlProvider for the D type invokes the addInlinedPath(...) method for the second inlined path.

6. The ObjectSqlProvider for the D type adds the oid of the d1.c field value to the INSERT statement.

7. The ObjectSqlProvider for the C type adds the oid of the d1.c.ai field value to the INSERT statement.

8. The ObjectSqlProvider for the A type adds the oid of the d1.b.ai field to the INSERT statement.

9. The PrimitiveSqlProvider for the int type adds the value of the d1.b.ai field to the INSERT statement.

Figure 5-6 - Example code and control flow showing the chain of responsibility for inserting a D instance.

Implementation Issues

In order to decrease the number of separate communications with the database, rather than each SqlProvider class executing its own INSERT and CREATE TABLE statements, it is passed a reference to an SqlBlock to which it can add SQL statements to be executed. When an object is passed to the persistence layer to be stored, a new SqlBlock is created and passed to the MasterSqlProvider's addInsertSql(...) method, which triggers the chain of responsibility necessary to add the relevant INSERT statements. Once all the necessary SQL commands have been added to the SqlBlock, they are sent to the database in a single message.

The algorithm followed by the addInsertSql(...) method of the ObjectSqlProvider is described in Figure 5-7. This is passed a TypeInfo instance by the MasterSqlProvider that contains the mapping metadata for the runtime type of the value. From this, the name of the table in which value should be persisted and the path values to be inserted are retrieved. Before adding the INSERT statement to the SqlBlock, a check is performed to see if the table exists. If the table does not exist then the addCreateSql(...) method is invoked to add a CREATE TABLE statement to the SqlBlock.
Detecting the existence of a table via JDBC is an expensive operation since it requires a `DatabaseMetaData` object to be constructed and returned. To avoid this for each table check, the database metadata is retrieved once on first connection to the database and then updated locally whenever a `CREATE TABLE` statement is executed. Unfortunately, this information is only accurate if there are no other processes modifying the database.

Since `CREATE TABLE` statements are not executed immediately, subsequent checks for that table must detect that the SQL for creating the table has already been added, otherwise it will be added multiple times which will cause an error when the SQL is executed. Therefore, the `SqlBlock` must maintain a list of table names for tables that it contains a `CREATE TABLE` statement for, but has not yet executed. When the `SqlBlock` is successfully executed, this list is added to the local cache of database metadata.

```java
public Long addInsertSql(SqlBlock block, 
                        Object value, 
                        TypeInfo typeInfo) {
    oid = generate_new_oid_for(value);
    tableName = typeInfo.getTableName()
    if (not block.tableExists(tableName):
        addCreateSql(block, typeInfo) // Adds the CREATE TABLE command to ‘block’
    command = new InsertCommand(tableName) // INSERT INTO <tableName>
    command.addValue("oid", oid) // Add the oid value for the “oid” column
    for (each path in typeInfo.getInlinedPaths()):
        addPathValues(block, command, path, value)
    block.add(command)
    return oid
}
```

```java
public Object getPathValue(SqlBlock block, 
                         InsertCommand command, 
                         PathTreeNode path, 
                         Object object) {
    field = path.getField() // Get the field corresponding to this node of the path
    value = next.getValue(object) // Get the value of that field from ‘object’
    valueOid = addInsertSql(block, value, get_type_info_for(value.getClass()))
    command.addValue(path.toString(), validOid)
    // Recursively call addPathValues on the relevant SqlProvider for each of the next
    // nodes in the path
    for (each node in path.getChildren():
        nextSqlProvider = masterSqlProvider.getSqlProviderFor(node);
        nextSqlProvider.addPathValues(block, command, node, value);
    }
}
```

Figure 5-7 - Pseudocode representation of the `addInsertSql(..)` method of the `ObjectSqlProvider`.

With this system, the oids of each field value in the path are inlined as well as the resulting path value. This ensures that each the oid of each object in the path is available when the object is reconstructed, which is necessary for tracking the object to ensure that reference semantics are preserved (discussed in Section 5.2.1.2). It also has the side-effect of allowing Foreign Key Aggregation to be expressed in terms of the path-inlining approach by adding each of the immediate fields of a class as inlined paths by default.
Although the design mentioned using a set of paths, where each path corresponds to a value to be added to the `INSERT` command, it is more efficient to use a path tree. In this tree, each node corresponds to a field declared in the type of its parent node’s field. Whereas before multiple paths may share the same initial fields, causing these paths to be traversed multiple times, the tree structure can represent these paths as having common parents so that each field is only visited once. This also prevents inlining intermediate oid fields multiple times and is vital for traversing paths involving array elements (discussed in Section 5.2.3).

5.2.1.2 Preserving Reference Semantics

In the persistence algorithm described so far, instances persisted in the database are, in essence, copies of their in-memory counterparts. Once an instance is persisted, there is no relationship between it and its persisted data, so it could potentially be persisted multiple times resulting in many copies in the database. To make matters worse, aliased objects would be persisted once-per-reference, with each reference pointing to a different copy of the object. This does not follow the reference semantics of the Java language.

To solve this problem, an `ObjectCache` class was introduced to maintain a mapping of in-memory object references to oid values. When an instance is first persisted, a reference to that instance is stored in the `ObjectCache` along with the generated oid value. If the same instance is subsequently passed to the persistence layer, the `ObjectCache` can be checked and, rather than inserting the instance again, its oid value can be used to update the instance’s row in the database. This feature is also used when reconstructing objects to ensure that two copies of an object are not made if the same object is requested twice. This ensures that observational equivalence is preserved within JVM executions, since in these situations the second request will return the exact same object as the first.

Implementation Issues

As mentioned in the prototype of this project (in Section 3.3), the `ObjectCache` must compare objects using pointer equality and only maintain weak references to them in order to avoid memory leaks. The Java standard API provides both a `WeakHashMap` class and an `IdentityHashMap` class to provide these two features separately, but not together. A `WeakIdentityHashMap` was constructed by extending the `WeakHashMap` implementation and overriding the comparison methods to perform a pointer comparison using `System.identityHashCode(..)`.

5.2.1.3 Naming conflicts

One of the problems with mapping objects from the Java domain to a relational database is that the range of valid names is different. In the PostgreSQL database, only the first 64 characters of an identifier are considered in comparison operations, but it is likely that fully-qualified class names in Java will be longer than this. If tables for storing class instances were named based on the fully-qualified name of the class, any classes belonging to a package with a 64-character name would map to the same table. Most relational databases also compare identifiers in a case-insensitive manner, so any Java classes that are identical except for case will map to the same table.

To solve these problems, a `TableNameMap` was introduced which stores a mapping between the fully-qualified name of a class and its corresponding table in the database. The default behaviour is to use the lowercase simple name of the class unless it clashes with an existing table, in which case a numeric suffix is added and incremented until no name conflict occurs. This mapping is stored to a

---

25 A fully-qualified class name is formed by prefixing the name of the class by the name of the package it belongs to.
special table in the database upon JVM termination so that it can be restored when the application is restarted.

Other naming conflicts are possible concerning the inlining of field paths. The column of the inlined path must be named by the full path, not just the final field in the path, in case multiple fields of the same type are inlined. In cases where this path is greater than 64 characters long, a disambiguation map must be used in the same way that table name conflicts are resolved.

5.2.1.4 Lazy Updating

One of the requirements described in Section 4.1.5 detailed the need to keep persistent data and in-memory data synchronised in order to maintain Java semantics. Following this strictly would require every data modification made to a persistent object to generate an UPDATE command to perform the same modification to its corresponding data in the database. In practice, since communication to the database server may be expensive, multiple consecutive updates should be grouped together in order to improve performance.

[Yeganeh 04] describes methods used to delay JDBC calls to improve performance and only execute them when necessary, based on some execution criterion. However, this is usually applied to SELECT statements, where the execution criterion evaluates the data dependency between the results of the JDBC calls and the Java code following them. Unfortunately, such a measure for determining when UPDATE statements should be executed is more difficult to find.

Since one of the aims of synchronisation is to ensure that the latest version of the object data survives even when the in-memory object is garbage collected, a potential synchronisation point for an object would be just prior to garbage collection. This could be achieved by inserting finaliser methods (discussed in Section 3.4.3.4) into persisted classes, however in certain situations finalisers may not be executed, may be executed multiple times, or may be executed a long time after the object became unreachable, which would cause unpredictable results.

For the purposes of a web application, another potential synchronisation point is upon the completion of a request. When a request is serviced, the relevant page class method may make alterations to persistent objects or persist new objects by assigning to its @Database-annotated fields. Each of these alterations must already be run in a single transaction to avoid the results of partially-executed requests from being stored, but delaying these changes until after the request has been fully serviced allows all the necessary updates to be executed in a single, short communication to the database. This reduces the time that locks on the affected database tables are kept, further improving performance.

As well as deciding when persisted data and in-memory data must be synchronised, another important issue is determining which parts of the data need to be synchronised. Although it is possible to compare the data already in the database with the in-memory data after a page class method has been run to detect which objects have been altered, this would likely produce worse performance than simply updating all the objects reachable from the method.

To improve performance, a method for tracking changes made to objects in the Java domain was needed. Both Aspect-Oriented Programming and bytecode manipulation tools were investigated for ways to solve this problem, and both approaches are described below for comparison.

Aspect-Oriented Programming

Aspect-Oriented Programming (AOP) [Kiczales et al. 96] allows developers to centralise code that needs to be duplicated across many different classes (termed cross-cutting concerns) and deploy this code either as a step between compilation and runtime or at the point where a class is loaded. The problem of determining which parts of an object have been altered can be seen as a cross-cutting concern, where we wish to inject code after every assignment to a field of a persisted object. This code
can then mark the object as ‘dirty’, indicating that its field values should be updated at the next synchronisation point.

**Figure 5-8** outlines a Java aspect that would accomplish the task of updating the database whenever the field of a persisted class \( A \) is altered. First, a pointcut is declared to capture all field modifications of \( A \) objects, followed by advice to be executed when such an event occurs. The advice checks that the receiver of the field modification is a persisted object, since both persistent and non-persistent instances of the same class may exist in memory. This check is performed by seeing if the object is in the ObjectCache (see Section 5.2.1.2). If the receiver is persistent, a call is made to the PersistenceManager to mark it as dirty.

```java
aspect LazyUpdateAspect {
    // Captures all assignments to fields
    pointcut fieldSet(obj): target(obj) && set(*,*);
    after(Object obj): fieldSet(obj) {
        // If the field was set on a persistent object, mark the object as dirty
        if (PersistenceManager.isPersistent(obj)) {
            PersistenceManager.markAsDirty(obj);
        }
    }
}
```

**Figure 5-8** - An example aspect for advising field assignments.

For an orthogonal persistence system, any type of object can be made persistent, and the set of types made persistent by an application cannot be determined in advance. To cope with this, it would be desirable to detect when the first instance of a type is made persistent and construct a custom aspect for detecting modifications to fields declared in that type. This aspect would then be woven into the rest of the application to catch these field modifications wherever they occur.

Unfortunately, aspects cannot be woven into classes that have already been loaded, so constructing custom aspects once a persistent type is known and weaving it on-the-fly is impossible. Instead, a single aspect must be woven at JVM initialisation which assumes that every type is persistent. The PersistenceManager can then perform the dynamic check to see whether the target of the field modification is persistent and mark it as dirty if appropriate. This approach adds the performance cost of checking whether an object is persistent to every field modification in the life of the application, and is therefore unacceptable for real-world use.

### Bytecode Analysis And Manipulation

The alternative approach to using aspects was to take advantage of Java’s Hot Code Replace feature, which allows classes to be modified at runtime such that all existing instances immediately exhibit the alterations. Although the hot swapping mechanism has been designed to support any binary-compatible alteration of a class, currently the Sun JVM only supports a very limited set of modifications. However, it is possible to change the bytecode of an existing method as long as the signature of the method remains the same.

This feature was leveraged for detecting modifications to fields of a type once an instance of that type has been persisted. At the point where the first instance of a class is persisted, the bytecode of the class is analysed and each method inspected in turn to determine which of the class’ declared fields are assigned in the method body. Each method is then modified to add a call to the PersistenceManager, notifying it of the fields that are assigned during that method so that they can be marked as dirty. Finally, the running class is replaced with the modified bytecode using the Hot Code Replace
mechanism. The details of this process are expanded further in the discussion of lazy loading in Section 5.3.1.4.

One problem with this method is that non-private fields may be modified directly by other classes, so all other classes in the web application would have to be analysed for these modifications to be detected. Since bytecode analysis is quite an expensive operation, lazy updates were restricted to just private fields to avoid this overhead. In practice, this does not have a large impact since most well-designed applications use private fields with getter and setter methods which will be lazily updated.

5.2.1.5 Inheritance

In Section 2.1.5.2, three methods for handling persistence of class hierarchies were suggested and compared. The specification of this project described the ability to apply each of these strategies to different parts of the same inheritance hierarchy, so that subtrees may be persisted in a manner that optimises performance based on their usage statistics. The first step to accomplishing this was to break each of the three methods of persisting class hierarchies into primitive operations between a class and its superclass. The following code will be used to demonstrate the different table layouts produced by applying each primitive operation.

```java
class A {
    int ai;
    public A(int ai) { this.ai = ai }
}

class B extends A {
    int bi;
    public B(int ai, int bi) { super(ai); this.bi = bi }
}

store(new A(56));
store(new B(72, 35));
```

![Figure 5-9](image_url) - Example code used to demonstrate the primitive inheritance operations.

Foreign Key Reference

The foreign key reference operation persists the subclass and superclass to separate tables, with rows corresponding to the same object being linked by a common oid value. Instances of the subclass have their subclass fields stored in the subclass table and their superclass fields stored in the superclass table. Instances of the superclass are only stored in the superclass table. Applying this operation to the entire tree is the same as applying the Table Per Class (TPC) strategy.

<table>
<thead>
<tr>
<th>A table</th>
<th>B table</th>
</tr>
</thead>
<tbody>
<tr>
<td>old</td>
<td>class</td>
</tr>
<tr>
<td>1</td>
<td>A</td>
</tr>
</tbody>
</table>

![Figure 5-10](image_url) - The result of the Foreign Key Reference operation.

Roll Up

The roll up operation moves all fields from the subclass table into the superclass table. All instances of the subclass are persisted to the superclass table. All instances of the superclass are persisted to the subclass table with NULL values for the remaining columns. Applying this operation to the entire tree is the same as applying the Table Per Inheritance Hierarchy (TPIH) strategy.
Copy Down

The copy down operation copies all fields from the superclass table into the subclass table. Instances of the superclass are persisted to the superclass table. Instances of the subclass are persisted to the subclass table along with their superclass fields. Applying this operation to the entire tree is the same as applying the Table Per Inheritance Path (TPIP) strategy.

Each of the arcs of the class hierarchy could then be assigned one of these primitive operations, allowing for combinations that produce hybrid hierarchy-table mappings. Unfortunately, this approach suffers from several drawbacks. Firstly, the resulting table mapping depends on the order in which each of the operations on the arcs of the class hierarchy is applied, so either a set order must be defined (such as breadth first from the root class), or the order encoded in the annotations of the arcs. This method also only allows these operations to be performed between a class and its direct superclass, whereas it may be desirable to apply different operations to the transitive arcs of the class hierarchy as well.

However, it is possible to eliminate these problems by likening inheritance at the database level to a reference between the subclass part of an object and its superclass, then expressing each of the operations using the path-based inlining technique found for persisting references. Each class in the hierarchy is now associated with a set of inlined paths that may refer to fields declared in superclasses or subclasses.

Implementation Issues

The path-based inlining approach as it stands only copes with persisting an object to a single table, however the Foreign Key Reference operation needs to persist the fields declared in each superclass to a different table. To cope with this, a TableInfo class was created to store a list of inlined paths per table and the TypeInfo class was modified to contain a list of TableInfo references. For insertion of an object, the addInsertSql(..) method was then altered to iterate through the TableInfo instances associated with the object’s type and produce a separate INSERT statement based on the inlined paths of each.

5.2.2 Storage Of Primitives And Autoboxed Primitives

Whereas the ObjectSqlProvider described in Section 5.2.1 is primarily concerned with persisting the structure of the data by storing oid references, the PrimitiveSqlProvider is responsible for storing each individual data item. Therefore, tables created by the PrimitiveSqlProvider
contain only two columns; the default oid column and a value column, whose type depends on the type of data being stored. Whenever a primitive is persisted, it is assigned an oid value and inserted into the table corresponding to its data type. Unlike the ObjectSqlProvider, there is no need for recursion, inlined paths or lazy updates, so this process is relatively simple.

Primitives in Java, even in their autoboxed form, can only be replaced rather than modified; they are immutable. Therefore, aliasing a primitive does not have as strong consequences as aliasing an object, since there is no way to modify the primitive such that it affects all other references to that primitive. This means that the measures put in place to ensure reference semantics are preserved for objects can be relaxed for primitives; they do not have to be put in the ObjectCache and, when inlined, they do not need to be separately persisted to their own table or assigned their own oid value.

**Implementation Issues**

Previously, the path-based inlining approach would persist each value in the path to its own and inline its oid before progressing to the next, since this allowed Foreign Key Aggregation and Single Table Aggregation to be handled by the same algorithm. To handle persistence of primitives, this was extended so that fields corresponding to references were annotated by an isOidInlined flag. When a reference field was reached in a path, the field value would only be recursively persisted and its isOidInlined if this flag is true, otherwise that field would simply delegate down to the next field in the path. A side-effect of this approach is that it can be applied to all immutable objects, not just primitives, simply by ensuring that all fields of immutable types are inlined and fields that correspond to immutable types have their isOidInlined flag set to false.

```java
class A { int ai; public A(int ai) { this.ai = ai; } }

// Each reference field in the path is parameterised by its isOidInlined value
// The following inlines the oid of the ai field and persists the ai value to the int table
setInlinedPaths(A.class, { A.ai(true) });
store(new A(25));

class B { int bi; public B(int bi) { this.bi = bi; } }

// The following inlines nothing – reference fields are merely placeholders if their isOidInlined value is false. The ai value is never persisted.
setInlinedPaths(B.class, { B.ai(false) });
store(new B(25));

class C { int ci; public C(int ci) { this.ci = ci } }

// The following inlines both the oid of the ai field and its value. The ai value is also persisted to the int table.
setInlinedPaths(C.class, { C.ci(true).value });
store(new C(25));

class D { int di; public D(int di) { this.di = di } }

// The following inlines only the ai value. It is not persisted elsewhere.
setInlinedPaths(C.class, { C.ci(false).value });
store(new C(25));
```
The definition of a path was also updated to distinguish between the oid of a primitive and the primitive value itself. The former is required when primitives are persisted on their own, while the latter is for use when primitives are referenced from other objects. To achieve this, fields of primitive type were altered to refer to the oid of the primitive and given a synthetic subfield called value, which represents the actual data stored in the primitive. **Figure 5-13** demonstrates the differences between these two concepts.

The example demonstrates that it is possible to specify a set of inline paths for an object that will omit important data, preventing the object from being fully reconstructed (as shown in the B class). Each SqlProvider is responsible for generating the default TypeInfo instance for the types it handles and ensuring that these are correct. It is then the responsibility of the heuristic optimiser to only perform operations upon the mapping that maintain all the paths necessary to persist the required object data.

### 5.2.3 Storage Of Arrays And Lists

Arrays and Lists contain sequences of objects that each share a common supertype, called the *component type*. In most situations, Lists are handled identically to arrays, except in the way that the component type is deduced. For arrays, the component type is attached to the dynamic type of the object and can be retrieved by calling `getClass().getComponentType()`. For Lists, the component type is not preserved at runtime, but may be deducible if the field containing the List instance is known by examining its generic type declaration (see Section 5.4.1). For the remainder of this section, the techniques applied to arrays can be assumed to apply to Lists except where explicitly mentioned.

Since the number of elements in the sequence is not known statically, it is not possible to store each array in a table column, so instead each element is stored in a separate row in the array’s table and all rows belonging to the same array share a common oid value. To preserve the ordering of elements, an explicit index column must be added which stores the index of each element within each array. Although it may seem that implicit ordering could be used, where the order in which rows are inserted and returned by the database reflects the order of the elements, an explicit index column allows individual elements to be easily identified in `UPDATE` or `DELETE` statements.

The most basic array tables will simply contain the oid of each of their elements, which can then be joined with the table corresponding to the component type when retrieving the element data. However, joining tables is an expensive process, and since all elements of an array share the component type as a common supertype, it should be possible to inline component type fields into the array table itself. Primitive arrays, such as `int[]`, are a prime example of when it would be inefficient to use a foreign-key lookup for each element, so moving the element value itself into the array table should increase performance. To achieve this, rather than having a single ‘array’ table for storing all arrays, different tables are created based on the component type. Each of these tables may have different inline paths corresponding to fields in their component type that should be inlined into the array table.

---

**Figure 5-13** - Example code and table layouts showing the different ways primitives can be persisted.
5.2.3.1 Array Inlining

When arrays are referenced from other objects, such as an array field in a class or a nested array, this will typically be translated into a foreign key reference between the referring table and the appropriate array table for the declared component type. In cases where the array and the referring object are always retrieved together, it would be more efficient to inline the array into the table of the referring class. To do this, the path inlining approach was extended to include a special ‘element’ path entry which refers to the elements of the array. From this, fields of the array’s component type can be inlined into the referring table (see Figure 5-14)

```java
class A {
    int ai;
    public A(int ai) { this.ai = ai; }
}
class B {
    int bi;
    A[] as;
    public B(int bi, A[] as) { this.bi = bi; this.as = as; }
}
```

// Possible paths to be inlined into the B table:
// 'B.as' inlines the oid of the A[] instance
// 'B.as.element' inlines the oids of each A instance in the array 'as'
// 'B.as.element.ai' inlines the oid of the 'ai' primitive for each A element
// 'B.as.element.ai.value' inlines the value of the 'ai' primitive for each A element
// element of the array 'as'

Figure 5-14 - Example code showing how array elements may be added to inlined paths.

In the above example, if the as.element path is inlined into the B table then each B object will no longer span a single row, but will contain one row for each element of as. This means it is no longer possible to simply generate a single INSERT statement for the referring object and fill in its inlined path values for each field, since if the path contains an array element then multiple INSERT statements may be needed. Each of these statements will contain duplicated data for the bi field of B, as shown in Figure 5-15.

```java
b = new B(7, new A[] { new A(2), new A(8), new A(5) });
setInlinedPaths(B.class, (B.as.element.ai.value));
store(b);
```

// Should generate the following SQL:

```sql
INSERT INTO B (oid, bi, as_index, as_element, as_element.ai_value) VALUES (1, 7, 0, 2, 2)
INSERT INTO B (oid, bi, as_index, as_element, as_element.ai_value) VALUES (1, 7, 1, 3, 8)
INSERT INTO B (oid, bi, as_index, as_element, as_element.ai_value) VALUES (1, 7, 2, 4, 5)
```

Figure 5-15 - Example code showing the multiple inserts needed to inline an array.

It is also worth noting that the index of each element should be implicitly inlined whenever an array element is inlined, since without this it is difficult to preserve the ordering of the array. This has the consequence that there is no longer a one-to-one mapping between elements of an inlined path and the resulting values of each INSERT statement.
5.2.3.2 Inlining Multiple and Nested Arrays

Intuitively, inlining array elements into a table should produce the same structure as a JOIN between the referring table and the array table. However, for multiple and nested arrays this results in generating the cross product of the elements of each array, which causes a large amount of duplication. It is possible to reduce the number of rows needed by sharing indices between arrays at the same level of nesting. Figure 5-16 through to Figure 5-18 demonstrate the differences between these approaches.

```java
class A {
    int[] ints1;
    int[] ints2;
    public A(int[] ints1, int[] ints2) { this.ints1 = ints1; this.ints2 = ints2; }
}
A a = new A(new int[] { 7, 9, 2 }, new int[] { 1, 4 });
setInlinedPaths(A.class, { A.ints1.element.value, A.ints2.element.value });
store(a);
```

<table>
<thead>
<tr>
<th>Cross product method</th>
<th>Shared indices method</th>
</tr>
</thead>
<tbody>
<tr>
<td>oid</td>
<td>ints index</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**Figure 5-16 - Example code and table layout for multiple arrays.**

```java
class A {
    int[][] ints;
    public A(int[][] ints) { this.ints = ints; }
}
A a = new A(new int[][] { new int[] { 7, 9, 2 }, new int[] { 5, 8 } });
setInlinedPaths(A.class, { A.ints.element.element.value });
store(a);
```

<table>
<thead>
<tr>
<th>Cross product method</th>
<th>Shared indices method</th>
</tr>
</thead>
<tbody>
<tr>
<td>oid</td>
<td>ints index</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**Figure 5-17 - Example code and table layout for nested arrays.**
Balancing simplicity and efficiency in Web applications

```java
class A {
    int[][] ints1;
    int[] ints2;
    public A(int[][] ints1, int[] ints2) { this.ints1 = ints1; this.ints2 = ints2; }
}

A a = new A(
    new int[][] {
        new int[] {7, 9, 2},
        new int[] {5, 8}
    },
    new int[] {1, 4}
);
setInlinedPaths(A.class, { A.ints1.element.element.value, A.ints2.element.value });
store(a);
```

**Figure 5-18** - Example code and table layout for multiple and nested arrays.

### Implementation Issues

Whereas before inlining a field required a single INSERT command to be passed between the SqlProviders corresponding to each node of the path tree, an inlined element node generates a separate INSERT statement for each element of the array. Each of these statements must duplicate any values that have already been added to the original INSERT and must eventually contain duplicates of any field values that have not yet been added, as demonstrated in Figure 5-19.

```java
class A {
    int ai1;
    int[] ais;
    int ai2;
    public A(int ai1, int[] ais, int ai2) { this.ai1 = ai1; this.ais = ais; this.ai2 = ai2; }
}

A a1 = new A(43, new int[] {7, 3, 8}, 21);
store(a1);
```
In the case of multiple inlined arrays at the same level of nesting, the first array will generate a separate INSERT statement for each of its elements, then the second array must edit these statements to add its own elements for the same indices. If the second array is shorter than the first, the remaining INSERT statements must be padded with NULL values. If the second array is longer than the first, new INSERT statements must be created with NULL values for the columns corresponding to the first array, as shown in Figure 5-20.

```java
class B {
    int[] bis1;
    int[] bis2;
    public B(int[] bis1, int[] bis2) { this.bis1 = bis1; this.bis2 = bis2; } }

B b1 = new B(new int[] { 9, 6 }, new int[] { 4, 8, 5 });
setInlinedPaths(B.class, ( B.bis1(true).element(false).value,
B.bis2(true).element(false).value ));
store(b1);
```
Balancing simplicity and efficiency in Web applications

Figure 5-20 - Sequence diagram showing how INSERT commands are constructed for multiple arrays at the same level of nesting (highlighted sections of SQL denote those added at that point in the sequence).

Nested inlined arrays pose a further problem, since they must duplicate the elements of the outer array that may then be modified by an array at the same level. To cope with these problems, a ContractedInsertCommand class was created which represents an abstraction of the SQL INSERT command in which nested field values and fields with multiple values are supported. A single instance of this class is passed between the SqlProviders involved in traversing the inlined paths and they each add a single tuple of field values to it. For ArraySqlProviders, a tuple containing a list of tuples is added, where each inner tuple corresponds to the field values to be inserted by each element of the array. Once the entire tree structure of field values has been added, an expand() method of ContractedInsertCommand traverses the tree and produces a sequence of INSERT statements to be executed on the database.

5.2.3.3 Empty and Null Arrays

Arrays with no elements are handled by inserting a single row into the array table containing just the oid value of the array and NULL values for the other columns. This row is needed to distinguish between an empty array and a null array reference, which would be indicated by a null oid value in the referring table. Arrays containing null references are persisted by inserting NULL values into the
element column. These can be distinguished from empty arrays because their index column will be non-null.

When persisting multiple arrays at the same level of nesting, it may seem wasteful to include a separate index column for each when all of the index columns for that level get incremented together, however these columns are useful for determining the lengths of each array. Since the number of rows inserted at each level of nesting is the length of the largest array at that level, smaller arrays must have some of their rows padded with NULL values and indices to match this length. Without a separate index column for that array, there would be no way to distinguish between null elements and padding rows.

5.2.3.4 Lazy Updating

To update the contents of an array, a separate UPDATE statement must be executed for each element, which can be an expensive operation if the array size is large. To improve upon this, assignment to particular array elements must be intercepted such that the array index can be marked as dirty and the UPDATE performed at the next synchronisation point. Unfortunately, array element assignments are as difficult to intercept as field assignments, and so are only really viable for private array fields.

However, updates to List elements are made through the List interface, which allows them to be easily intercepted by inserting code into those methods in the particular List implementation being persisted. For the purposes of this project, only the ArrayList implementation was considered; additional methods may need to be advised for other implementations if they do not build upon the methods provided by the List interface.

Unlike arrays, lists can grow and shrink dynamically through insertion and deletion of elements. If the list shrinks, the rows corresponding to the end elements can be removed by a single DELETE command. In this case, it does not matter which elements were deleted, so this is represented by setting a special dirty flag which corresponds to the entire list. At the next synchronisation point, if this flag is set, then a DELETE command is issued for all rows of that list greater than the last index of the list.

If elements are inserted or have their indices altered then a separate UPDATE command must be issued for each to update their corresponding rows in the ArrayList table. This is represented by setting a dirty flag for the index of each affected element, which will each generate a single UPDATE command to refresh the row corresponding to that index.

The ArrayList class contains the following methods that are intercepted so that lazy updating can be performed:

- **public E set(int index, E element)**
  Sets the element at the given index to element. A method call to the PersistenceManager is inserted to mark the row at index as dirty. This will generate an UPDATE command for that the row corresponding to that index which will cause element to be persisted if it does not already exist in the database.

- **public boolean add(E o)**
  Adds o to the end of the list. This is handled identically to a call to the set method where the index is the current length of the list.

- **public void add(int index, E element)**
  Inserts element at the given index in the list. All elements at or after index have their indices incremented. A method call to the PersistenceManager is inserted which marks all indices from index up to the size of the list as dirty. This will cause an UPDATE command to be executed for each of those indices.
• public E remove(int index)
  Removes the element at the given index. This is handled identically to the add method, but additionally the dirty flag for the list is set to true.

• public boolean remove(Object o)
  Removes any instances of the given object from the list. This method iterates through the list to find the indices that correspond to o and then calls a fastRemove(int index) method on each. This is handled by intercepting at the fastRemove(..) method in the same way that the remove(..) method is handled.

• public void clear()
  Removes all elements from the list. A method call to the PersistenceManager is inserted which sets the dirty flag of the list to true.

5.2.4 Storage Of Sets And Maps

Although Sets do not require the ordering of their elements to be preserved, it was not possible to eliminate the index column completely since it is required to distinguish between the end of a Set and a null Set element. In the end, the strategy for persisting arrays was found to be the most efficient implementation for persisting Sets. Maps are also persisted in a similar way, except that each row contains a reference to both the key and value objects for that element. For inlined paths, a new key path entry was added which allows paths through the key object to be inlined.

5.3 Object Retrieval

This section concentrates on the design and implementation of the classes concerned with retrieving objects from the database. When an object is requested from the persistence layer, first the SQL for fetching the necessary data is assembled, and then once the data has been retrieved the required objects are reconstructed and returned. Each of the following subsections describes how these tasks are achieved for each of the required data type groups.

5.3.1 Retrieval of Class Instances

Once an object is stored in the database, its data and the data of its fields are potentially distributed and duplicated over a large number of tables. It is the task of the retrieval algorithm to take a request for an object and piece together the required data from the database in an efficient manner. At a basic level, this can be seen as a recursive process, whereby first all of the field values are retrieved, then the parent object is reconstructed and the references to its field values assigned. Unfortunately this process is complicated by the path-inlining strategy (see Section 5.2.1.1), in that the immediate fields of an object may need to be fetched from different tables based on the context from which they are accessed.

To efficiently reconstruct an object, it is useful to leverage information available about the object prior to accessing the database. Since the persistence system works by storing values of fields marked with the @Database annotation, the input to retrieving such a field value will be the declaration of the field, including its name and type. Together, these are used to uniquely identify an object in the database, and the corresponding oid of the object can be obtained via a field lookup table. The type of the field can be used further to optimise the retrieval process, since it gives an upper-bound on the class of the object being fetched.
5.3.1.1 Field References

A simple strategy for fetching object data would be to assume that each field value is represented by a single row that has been stored in a table corresponding to the static type of the field. First, a `SELECT` statement is executed on that table to retrieve the row corresponding to the field’s oid value. The columns of this row contain the oids of each subfield of the object, and the types of each subfield can be obtained via reflection on the class of the object, so the retrieval process can be called recursively with the oid and type of each subfield to retrieve each of the subfield values. This method works, but each object fetched requires a separate `SELECT` statement to be constructed, executed and then its results analysed before retrieving its field values, as shown in Figure 5-21.

```java
class A {
    int a1;
}
class B {
    int bi;
    A a1;
    A a2;
}
B b = retrieve(1, B.class);
```

<table>
<thead>
<tr>
<th>oid</th>
<th>class</th>
<th>bi</th>
<th>a1</th>
<th>a2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>B</td>
<td>2</td>
<td>3</td>
<td>5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>oid</th>
<th>class</th>
<th>a1</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>A</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>A</td>
<td>6</td>
</tr>
</tbody>
</table>

// Bracketing denotes the contents of individual requests to the database
```sql
{ SELECT B.bi, B.a1, B.a2 FROM B WHERE B.oid = 1 }
{ SELECT INT.value FROM int WHERE int.oid = 2 }
{ SELECT A.a1 FROM A WHERE A.oid = 3 }
{ SELECT INT.value FROM int WHERE int.oid = 4 }
{ SELECT A.a1 FROM A WHERE A.oid = 5 }
{ SELECT INT.value FROM int WHERE int.oid = 6 }
```

**Figure 5-21** - Example code, table contents and SQL for the basic object fetching algorithm.

A more efficient strategy would minimise the number of separate communications with the database by combining the individual statements into a single `SELECT` that retrieves all the necessary data from the three tables. In this case, all the data will be returned in a single result table, so the structure of this table must be carefully designed to allow the data for each separate object to be easily extracted. This is done by naming each column with the data path it represents, an example of which is shown in Figure 5-22.

**Figure 5-22** - Table layout of the result table for B objects.

<table>
<thead>
<tr>
<th>oid</th>
<th>class</th>
<th>bi</th>
<th>bi value</th>
<th>a1</th>
<th>a1 class</th>
<th>a1 ai</th>
<th>a1 ai value</th>
<th>a2</th>
<th>a2 class</th>
<th>a2 ai</th>
<th>a2 ai value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>B</td>
<td>2</td>
<td>57</td>
<td>3</td>
<td>A</td>
<td>4</td>
<td>32</td>
<td>5</td>
<td>A</td>
<td>6</td>
<td>34</td>
</tr>
</tbody>
</table>

With the result table layout fixed, the problem is now to construct an SQL statement that produces a table of that format given the distribution of the data specified in the object-relational mapping. For traversing fields persisted using a foreign-key reference, this may be performed either by a series of nested `SELECTs` or `JOINs`.  

111
Balancing simplicity and efficiency in Web applications

Nested Select Approach

For the nested SELECTs approach, the final result table is viewed as a SELECT over the result tables for constructing the individual sub-fields of the object, which in turn are SELECTs of their sub-fields. With this design it is possible to build up the complete SELECT statement recursively, first by drilling down to the leaves of the tree and constructing their SELECT tables, then by adding further levels of nested SELECTs for each layer of references by unravelling the recursion. With this approach, the outer SELECT is over the table corresponding to the type of object being reconstructed and contains the WHERE clause for identifying the particular object required.

![Figure 5-23 - Table layouts of the result tables for ints and A objects.](image)

<table>
<thead>
<tr>
<th>oid</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>57</td>
</tr>
<tr>
<td>4</td>
<td>32</td>
</tr>
<tr>
<td>6</td>
<td>34</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>oid</th>
<th>class</th>
<th>ai</th>
<th>ai_value</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>A</td>
<td>4</td>
<td>32</td>
</tr>
<tr>
<td>5</td>
<td>A</td>
<td>6</td>
<td>34</td>
</tr>
</tbody>
</table>

Sql commands generated:

1. SELECT int.oid, int.value FROM int
2. SELECT int.oid, int.value FROM int
3. SELECT A.oid, A.class, A.ai, A_ai.value AS ai_value FROM {
   SELECT int.oid, int.value FROM int
} AS A_ai, A WHERE A.ai = A.ai.oid
4. SELECT B.oid, B.class, B.bi, B_bi.value AS bi_value, B.ai, B_a1.ai AS a1_ai, B_a1.ai_value AS a1_ai_value, B.a2, B_a2.ai AS a2_ai, B_a2.ai_value AS a2_ai_value FROM {
   SELECT int.oid, int.value FROM int
} AS B_bi, {
   SELECT A.oid, A.class, A.ai, A.ai.value AS ai_value FROM {
   SELECT int.oid, int.value FROM int
} AS A_ai, A WHERE A.ai = A.ai.oid
} AS B_a1, {
   SELECT A.oid, A.class, A.ai, A.ai.value AS ai_value FROM {
   SELECT int.oid, int.value FROM int
} AS A_ai, A WHERE A.ai = A.ai.oid
} AS B_a2, B WHERE B.bi = B.bi.oid AND B.ai = B_a1.ai AND B.a1 = B_a1.ai.oid AND B.a2 = B_a2.ai AND B.oid = 1

![Figure 5-24 - Sequence diagram showing how nested SELECT commands are constructed for fetching B instances (highlighted sections of SQL denote those added at that point in the sequence).](image)

It is worth noting that the SELECT command for the B table contains two nested SELECTs which are identical but named differently (B_a1 and B_a2). This is necessary since the B table references two A objects that each map to different columns in the resulting table, so the renaming is required to be able to distinguish between the two subtables. Another side-effect of the recursive process is that the column names of the result tables get renamed at each level. This could be reduced by propagating the full path
of each field down through the recursion such that each column of the result table is only ever renamed once, however most DBMSs do not follow the process described by the SQL literally and are likely to eliminate the intermediate renaming steps anyway.

**Join Approach**

The alternative method of building up the result table is to use nested JOIN syntax, where each reference traversed becomes a JOIN between the referring and referenced tables. With this approach it is only possible to rename tables within each JOIN and not individual columns, so all column renaming must be performed at the outermost level, in the SELECT part of the query. Figure 5-26 shows the SQL produced by this approach.

### Result table for fetching B objects

<table>
<thead>
<tr>
<th>old</th>
<th>class</th>
<th>bi</th>
<th>bi_value</th>
<th>a1</th>
<th>a1_class</th>
<th>a1_ai</th>
<th>a1_ai_value</th>
<th>a2</th>
<th>a2_class</th>
<th>a2_ai</th>
<th>a2_ai_value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>B</td>
<td>2</td>
<td>57</td>
<td>A</td>
<td>4</td>
<td>32</td>
<td></td>
<td>A</td>
<td>6</td>
<td>34</td>
<td></td>
</tr>
</tbody>
</table>

**Figure 5-25 - Table layout of the result table for B objects.**

Sql commands generated:

1. SELECT int.oid, int.value FROM int
2. SELECT int.oid, int.value FROM int
3. SELECT int.oid, int.value FROM int
4. SELECT A.oid, A.class, A.ai, A_ai.value AS ai_value FROM A JOIN (int AS A_ai) ON A.ai = A_ai.oid
5. SELECT B.oid, B.class, B.bi, B_bi.value AS bi_value, B.ai, B_a1.class AS a1_class, B_a1.ai AS a1_ai, B_a1_ai.value AS a1_ai_value, B_a2.class AS a2_class, B_a2.ai AS a2_ai, B_a2_ai.value AS a2_ai_value FROM {
   (B JOIN int B_bi ON B.bi = B_bi.oid)
   JOIN (A B_a1 JOIN int B_a1_ai ON B_a1.ai = B_a1_ai.oid)
   ON B.a1 = B_a1.oid)
   JOIN {
   A B_a2 JOIN int B_a2_ai ON B_a2.ai = B_a2_ai.oid
   ) ON B.a2 = B_a2.oid
   ) WHERE B.oid = 1

**Figure 5-26 - Sequence diagram showing how SELECT commands with JOINs are constructed for fetching B instances (highlighted sections of SQL denote those added at that point in the sequence).**

Since JOINs are commutative and associative, it is possible to flatten the nested JOINs to produce the SQL shown in Figure 5-27, which simplifies the SQL generation algorithm. With this approach, the process for reconstructing an object would take a SELECT statement and add columns for each of its fields, the table that the object’s row resides in and filter expressions for finding the rows corresponding to each field value. As before, this is complicated by the need to uniquely name each table involved, which can be done by propagating the path to the data item down the recursion.
Both the nested SELECT and JOIN approaches were tested on a PostgreSQL database filled with 10,000 rows for each of the B, A and int tables. The plan for executing each was then inspected and found to be identical (shown in Figure 5-28), demonstrating that the PostgreSQL planner automatically converts nested SELECTs to JOINs and reorders the JOIN operations to improve efficiency. However, using nested SELECTs increases the search space of possible plans that the PostgreSQL planner has to consider, so there is a limit to the number of nested SELECT statements that will be automatically collapsed. Since the WHERE clause to restrict the number of rows searched is in the outer-most SELECT, not collapsing the inner selects would likely cause a big performance hit as those tables would be fully traversed. Therefore, to eliminate this possibility, the JOIN approach is always used in preference to nested SELECTs.
Reconstruction Of Objects

Once the required object data has been fetched from the database, the requested object must be instantiated and each of its fields filled with the relevant data, which is done by recursively reconstructing the objects referenced by each field. The columns of the data table are named based on the path to the value that they contain, so the path to each field is passed to the recursion so that each SqlProvider knows which columns of data should be read for reconstructing their objects.

5.3.1.2 Null References

Although it would seem obvious to persist null values for fields as NULL values in the foreign key columns corresponding to the oid of the field value, this choice causes several problems when generating efficient queries. The SQL JOIN operation will only return rows for which there is a match in both the joined tables, so performing a JOIN between two tables where the foreign key value is NULL will produce no matches. This is undesirable because it means that a SELECT over an object graph which contains at least one null reference will always produce no data, even when some non-null references exist.

![Figure 5-29 - Plan produced by PostgreSQL for LEFT OUTER JOIN approach.](image)

It is possible to alter the generated SQL to use a LEFT OUTER JOIN in order to avoid this problem. This operation produces the UNION of a normal (inner) JOIN and any rows of the left operand for which there are no matches in the right operand. To make the rows of the left operand union-compatible with the result of the JOIN, the remaining columns are filled with NULL values, as required. Unfortunately, the LEFT OUTER JOIN operation is neither commutative nor associative, so it cannot be flattened to a simple SELECT statement. This also restricts the options for planning how to efficiently execute the query, since each JOIN cannot be reordered to reduce the number of rows that need to be scanned at each level. Figure 5-29 shows the plan produced for the same query used in Figure 5-28 except using LEFT OUTER JOINS in the place of the inner JOINs. This can severely

---

26 This occurs not only because the oid field of the referenced table is a primary key field and so cannot be NULL, but also because two NULLs are not considered equivalent.
affects the performance of selecting a single object depending on the number of rows in the database, as shown in figure **Figure 5-30**.

![Performance difference between JOIN types](image)

**Figure 5-30** - Graph of performance difference between JOIN types.

To overcome this problem, instead of using NULL values to represent null references, the oid value zero is used. Whenever a table is created, a new row is automatically inserted which has a zero oid value and zero for all foreign key oid columns. Value columns (such as primitive values or array element indices) are filled with NULL values as before, since these are not included in the JOIN conditions.

### 5.3.1.3 Inlined Paths

So far, the SQL generation algorithm has assumed that all field values are stored in separate tables, however one of the main features of the object storage algorithm was that it was capable of inlining values of paths that are commonly accessed. The current method of each object recursively calling the object retrieval process to reconstruct its fields must now be altered such that the object knows in advance which paths of its fields are to be traversed. It can then compare this set of required paths with the set of inlined paths for the table corresponding to its type in order to work out which data can be fetched directly from that table and which data must be fetched using a JOIN.

### Implementation Issues

To handle inlined paths, the process of retrieving an object was split into two stages; first, building a tree of paths to fields that must be retrieved from the database, then second, traversing this tree to generate the parts of the SELECT statement for fetching the field data. Building the tree of required paths is achieved by a recursive search through the static object graph, which filters out fields that aren’t persisted (i.e. those declared as static, final or transient) or for which the data can be loaded lazily (i.e. those declared as private). Once this has been constructed, it can be compared to the inlined paths of a table to determine which table the required data should be fetched from. The intersection between the two path trees indicates the required path values that are inlined and so can be
fetched from the current table, while the difference indicates which of the required path data exist in a separate table and require a JOIN to be added to the SELECT command.

Once the data has been fetched from the database, the required path tree is traversed again to reconstruct the objects for each field. Any fields of an object for which there is no data are left alone, since these will either be loaded lazily or should be left at their initialised values. This strategy is complicated by inheritance, since once the type of an object is known, further field values may need to be fetched from the database. This is discussed in detail later on in Section 5.3.1.5.

5.3.1.4 Lazy Loading

Objects can be reconstructed lazily if accesses to the fields of persisted objects can be detected and intercepted by the persistence system so that it can retrieve and reconstruct the field value immediately before it is required by the web application. This provides the benefit that object data is only loaded when needed, eliminating the unnecessary data fetched by eager loading.

The approach used to achieve this is very similar to the one used for lazy updates, described in Section 5.2.1.4. When the first instance of a class is persisted, the class is put through an enhancement phase where all the private fields are found and their usage within methods determined. Code is then inserted at the top of each method to fetch the values of those fields before the rest of the method is executed.

Implementation Issues

Whenever an instance of an enhanced class is retrieved from the database, the values of the private fields are not fetched but instead left as null. Whenever any of the methods that access private fields are called, they invoke the fetchFields method of the SqlProvider which fetches the necessary values from the database and assigns them to the fields. The first task of the SqlProvider is to look up the passed instance in the ObjectCache to determine whether it is persistent or not. If the instance is persistent, its oid is obtained and all of the required field values required by the method are requested in a single SELECT statement and assigned to the fields of the instance using reflection.
Balancing simplicity and efficiency in Web applications

One of the side-effects of lazy loading is that field data is often retrieved from within the class in which the field was declared. Referring to the classes in Figure 5-32, even though all the data from the A class is inlined into the B table, when the b.a.getAi() method is called, the A instance has no knowledge of the referring B, and so does not know to look in the B table for the field data. The consequence of this is that the A data must be inserted into both the A table and the B table, so that all A data resides in a known place.

```java
class A {
    private int ai;
    public A(int ai) { this.ai = ai; }
    public int getAi() { return ai; }
}
class B {
    public B(A a) { this.a = a; }
}
B b = new B(new A(7));
setInlinedPaths(B.class, { B.a.ai });
long oid = store(b);
--- JVM restart ---
B b = retrieve(oid);
b.a.getAi(); // Attempts to fetch the value '7' for field 'A.ai'
```

Figure 5-32 - Example demonstrating the side-effects of lazy loading.

5.3.1.5 Inheritance

So far, the object retrieval and reconstruction algorithm has always been able to fetch all object data in a single SELECT query, albeit with multiple JOINs across the required class tables. However, with inheritance, the runtime type of an object may not necessarily match the static type of the fields referencing it. This causes big problems since the static type of each field is used to determine which tables to join to fetch that field’s value.

For the Table Per Class and Table Per Inheritance Hierarchy approaches, fetching an object of runtime type t from the table of any supertype of t will still produce some data for that instance from which the runtime type of the object can be found. However, the Table Per Inheritance Path approach persists an instance of a class to the class’ table without also storing data in the superclass tables, so querying across a superclass table will produce no data for that instance. To avoid this, queries over superclass tables must also be UNIONed with queries over all tables for subclasses persisted using the Table Per Inheritance Path technique.

Once the runtime type of an object is known, additional data may be required to reconstruct that type if it is a subclass of the static type of the field it was referenced from. To account for this, a separate SELECT statement must now be executed to fetch the field values for any persistable fields declared in the classes between the runtime type and the static type in the class hierarchy. It is also worth noting that those field values themselves may differ from the declared types of their fields, so two queries may have to be executed for each reconstructed object in the worst case.

Implementation Issues

The addition of further SELECTs in the reconstruction process required a fairly substantial change in the design of the persistence layer. Without inheritance, each SqlProvider would simply construct and execute some SQL and then build up each object based on the result; there was no need for any state information to be recorded about each object, so one SqlProvider instance for each type
The Persistence Layer

sufficed. With inheritance, it is now possible to have a partially reconstructed object that is waiting for further data. Therefore, each object must have a corresponding SqlProvider that keeps state information about the fetched fields of the object and produces further SELECTs as necessary.

Since retrieving an object is the only operation where state for each object must be maintained, the existing SqlProvider classes were left alone for construction of INSERT, CREATE TABLE and UPDATE commands. A new set of classes, called SqlProducers, were written for handling the retrieval of individual objects. When an object request is first received, one SqlProducer is instantiated for each field value in the object graph corresponding to the object’s static type information, mirroring the expected structure of the reconstructed object graph. This graph of SqlProducers is then traversed and their addSelectSql(…) methods invoked to construct a single SELECT statement needed to query the tables corresponding to the static type of each object.

```java
class A {
    int ai;
    public A(int ai) { this.ai = ai; }
}
class B extends A {
    int bi;
    public B(int ai, int bi) { super(ai); this.bi = bi; }
}
class C {
    A a;
    public C(A a) { this.a = a; }
}
C c1 = new C(new A(45));
C c2 = new C(new B(32, 21));
Long c1_oid = store(c1);
Long c2_oid = store(c2);
```

Figure 5-33 - Example code and table layouts for inheritance persistence using the Table Per Class strategy.

Once the first SELECT has been executed, the result data is propagated through the graph of SqlProducers, allowing each to find the runtime type of its object and instantiate it. If the runtime type of each object in the graph corresponds to the static type then the reconstruction phase is complete and the requested object can be returned. Otherwise, the addSelectSql(…) method is called again for those SqlProducers that require additional data and a set of SELECT commands is constructed. Since there may be multiple SqlProducers that require additional data from different parts of the graph, each constructs a separate SELECT command and adds it to the SqlBlock. All of these commands are then executed together, so each command has a reference to the SqlProducer that generated it so that the returned result sets can be passed to the correct one. The process of gather SELECT statements and fetching data continues until every SqlProducer in the tree has all the data needed to reconstruct its object, at which point the requested object is returned.
Figure 5-34 - Sequence diagram showing the reconstruction of a C object with an A field value.
5.3.2 Retrieval Of Primitives

Primitive types cannot be subclassed, so therefore their retrieval process is relatively simple. Each PrimitiveSqlProducer will always produce exactly one SELECT statement which will return the required value.
5.3.3 Retrieval Of Arrays And Lists

The strategy for persisting arrays described in Section 5.2.3 produces an array results table that introduces a number of problems into the reconstruction algorithm. Unlike reconstruction of objects that only had to handle a single row in the results table, results for SELECTs involving array tables contain multiple rows that must be distributed to the correct SqlProducer instances. Arrays on the same level of nesting share their indices, so each row must be distributed to each of their SqlProducers. However, for arrays on different levels of nesting each row corresponding to an element of the inner array contains duplicated data for the outer array, therefore only the rows corresponding to the first element of each inner array should be passed to the SqlProducers for the outer array.

Implementation Issues

Access to multiple result rows is provided by the JDBC interface in the form of a next() method in the ResultSet class which allows the current row pointer to be incremented. This means that data can only be accessed in a sequential fashion; there is no way to traverse backwards through the results, except to cache the values retrieved from previously-accessed rows. One of the main problems with implementing array reconstruction was ensuring that every SqlProducer had extracted all the required data from the current row before incrementing the row pointer.

Since multiple arrays on the same level of nesting require data from the same rows, the next row pointer can only be incremented after all SqlProducers on that level have read all their required values. This was difficult to achieve because multiple nested arrays occur in separate areas of the object graph, but their SqlProducers must communicate with each other in order to decide when the row pointer can be incremented. To achieve this, the root SqlProducer that receives the initial object request is assigned the task incrementing the row pointer where appropriate, since it is the only node in the graph that is guaranteed to have paths to all other nodes.

```java
private void propagateArrayData(IDataset data) {
    build a set of ArraySqlProviders at the lowest level of nesting and pass it to
    propagateArrayData(..)
}

private void propagateArrayData(IDataset data, Set<ArraySqlProvider> sqlProviders) {
    do:
        for each arraySqlProvider in sqlProviders:
            pass the current row of data to arraySqlProvider
            construct a childSqlProviders set containing all ArraySqlProviders at a level of
            nesting deeper than those in sqlProviders
            if childSqlProviders is empty:
                increment the current row pointer
            else:
                recursively call propagateArrayData with childSqlProviders
    until all elements of all sqlProviders have been reconstructed or the end of the
    data is reached
}
```
Figure 5-36 - Pseudocode algorithm for distribution of row data to multiple and nested ArraySqlProducers.

Figure 5-36 contains a pseudocode representation of the algorithm used by the root SqlProducer to distribute row data to each ArraySqlProducer as required. One of the requirements for the algorithm to work is that each ArraySqlProducer can detect when it has reached the last element to be reconstructed. Unfortunately, this is more difficult than it might seem, since in nested arrays, the data of each inner array uses the same columns, as shown in Figure 5-37.

store(new int[][] { new int[] { 53, 49, 74 }, new int[] { 12, 46 } });

<table>
<thead>
<tr>
<th>old</th>
<th>index</th>
<th>element</th>
<th>element_index</th>
<th>element_element_value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>53</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>3</td>
<td>1</td>
<td>49</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>4</td>
<td>2</td>
<td>74</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>5</td>
<td>0</td>
<td>12</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>6</td>
<td>1</td>
<td>46</td>
</tr>
</tbody>
</table>

Figure 5-37 - Example code and table layout for a nested int array.

To detect when the inner most array has reconstructed all of its elements, its ArraySqlProducer has to keep track of the last value read in its index column. If the next row contains an index value other than the successor of the previous value, then the end of that array has been reached. It is worth noting that the index column may also contain NULL values in the case where multiple arrays of different lengths exist at the same level of nesting.

5.3.4 Retrieval Of Sets And Maps

Since Sets and Maps are persisted in the same way as arrays and Lists, the same implementation was used to fetch their data and reconstruct their elements. For reconstruction of Maps, both the key and value objects are reconstructed for each element.

5.4 Optimisations

Having implemented a flexible persistence system, the focus of the project switched to optimising the queries and table schemas used in order to improve the overall performance of the web application. Firstly, an algorithm was implemented for propagating generic types through class hierarchies so that more static type information would be available for optimising queries over parameterised class instances. A heuristic optimiser component was then created to gather usage statistics for persistent objects and adjust the object-relational mapping to improve performance for these usage patterns. This section describes the design and implementation of both of these features.

5.4.1 Resolving Generic Types

Generic type declarations contain extra information about the allowed types of the values they contain, which can be used to improve the performance of storing and fetching those values. However, the generic type of an object is not preserved at runtime, so cannot be obtained by reflecting on the object’s runtime type. Fortunately, it is possible to obtain static type information by reflecting on a field that references the object, as shown in Figure 5-38.
class A<T> {
    T t;
}

class B {
    A<Integer> a = new A<Integer>();
    void test() {
        // prints "A" since generic types of objects are not preserved at runtime
        System.out.println(a.getClass());
        // prints "A<Integer>" since generic types of fields are preserved
        System.out.println(B.class.getDeclaredField("a").getType());
    }
}

Figure 5-38 - Java code demonstrating using field reflection to obtain generic type information.

The main motivating example for using generic type information is to allow Java Collections to be handled in the same way as arrays. If an Integer[] instance is stored, it is placed in the Integer[] table which contains the Integer values inline, rather than referenced via foreign keys. This prevents the considerable performance hit of persisting the array and its elements to separate tables and using a JOIN when the array needs to be retrieved, but is only possible because the array may only ever contain Integers which are immutable objects. Since the component type of the array is known, a table may be created that is customised to that type, allowing such optimisations to occur.

<table>
<thead>
<tr>
<th>java.lang.Integer[] table</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>oid</strong></td>
</tr>
<tr>
<td>---</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>1</td>
</tr>
</tbody>
</table>

Figure 5-39 - Example layout of table for storing Integer[] instances.

This can be compared to the persistence of an ArrayList with no generic information. Even though it would be possible (albeit highly inefficient) to iterate through the list and find that it contains only Integer instances, there are no guarantees that it will only ever contain Integers. Persisting this ArrayList to a specialised ArrayList<Integer> table would fail if, for instance, a String was then added to it. Given this restriction, the persistence system could default to storing the object in a general ArrayList table which contains the runtime type name and oid of each element. Obviously, for a list that only contains Integers, persisting each element to a separate table with a foreign key reference would almost double the storage requirements and severely impact the performance of retrieval due to the need to JOIN the two tables.

<table>
<thead>
<tr>
<th>java.util.ArrayList table</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>oid</strong></td>
</tr>
<tr>
<td>---</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>java.lang.Integer table</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>oid</strong></td>
</tr>
<tr>
<td>---</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>7</td>
</tr>
</tbody>
</table>

Figure 5-40 - Example layout of tables for storing an ArrayList of Integers.
Unfortunately, determining the generic type of an object is complicated by inheritance, since obtaining the generic type information from a field declaration only gives the static type of the object which may be a superclass or superinterface of the dynamic (runtime) type. In the example code in Figure 5-41, although the generic type information for the B class is known, this must be propagated to the superclass to determine the type of A.x and to the subclass to determine the type of C.x.

```java
class A<X> { 
    X x;
}
class B<X, Y> extends A<Y> { 
    int bi;
}
class C<X> extends B<X, Float> { 
    X x;
}

// Create a new C such that A.x is of type Float and C.x is of type Integer
B<Integer, Float> b = new C<Integer>();
```

**Figure 5-41 - Code example demonstrating problems caused by generics and inheritance.**

It is worth noting that the Java Language Specification forbids a class to implement an interface twice with different generic parameters anywhere in its supertype hierarchy. Therefore, for any type t for which generic type information is known, it is possible to propagate this information up to every superclass and superinterface of t and find a single mapping between each type variable and its corresponding type. For subclasses, it is possible to trace the mapping of type variables to types down the hierarchy, but this will not include the types of variables introduced in subclasses that are not propagated up to their supertypes. Figure 5-42 shows some example cases where this strategy succeeds and fails.

```java
class A<X> { }
class B<X, Y> extends A<Y> { }
class C<X, Y, Z> extends B<Y, Z> { }

// Given B<Integer, Float>, it is possible to resolve:
// C<?, Integer, Float> extends B<Integer, Float> extends A<Float>
```

**Figure 5-42 - Code example showing generic type variable propagation.**

### Implementation Issues

The goal of the generic type propagation algorithm is to determine the type of a generic type variable for a class (the target class) given generic type information about one of its super or subclasses (the known class). For the purposes of implementation, this was split into the following cases:

- The known class and the target class are the same. In this case, reflection is used to look up the type of the named type variable in the known class.

- The known class is a subclass of the target class. In this case, a recursive process is used to propagate the generic type mapping from the known subclass to each superclass until the target class is reached.

- The known class is a superclass of the target class. In this case, a search must be performed to find the path of classes connecting the known and target classes in the class hierarchy. Once this has been done, a recursive process is used to propagate the generic type mapping down from the known class to each subclass in the path until the target class is reached.
The last case is complicated by the fact that the known class may be an interface, so the algorithm for finding the path of classes connecting the known and target classes had to account for multiple inheritance. Unfortunately, this increases the search space and makes the algorithm quite inefficient for finding large paths. A potential optimisation would be to construct the mapping for the target class in terms of the known class variables so that future lookups between the same classes can be performed without executing the algorithm.

5.4.2 Optimisation Heuristics

Having implemented a system that can inline field paths in order to improve performance, a separate component was implemented which attempts to find combinations of inlined paths that give the best performance. To achieve this, the persistence algorithm was first altered to record a set of statistics concerning the field accesses and updates of each object, and then a cost function based on this heuristics was used to grade the performance of a particular inlining configuration given these access patterns.

The aim of the cost function is to estimate the average storage and retrieval time of the persistence system given a particular inlining configuration and a set of statistics of object usage. This is calculated as a sum of the average cost of retrieving and updating each persistent type in the database. Deletion was not considered since it is performed by a separate garbage collection algorithm and does not directly affect the performance of fulfilling page requests. Insertions were also not included since inlining of fields only causes a negligible change in their execution time.

```java
class A { int ai; }
class B { A a; }
class C { A a; }
```

*Figure 5-43 - Example classes used to demonstrate the heuristic cost function.*

Given the classes in Figure 5-43, the average cost of each class is computed by summing its average selection, insertion and update costs, which in turn depend on the costs of the sub-fields. For Foreign Key Aggregation, selection of each object requires an oid lookup, which in PostgreSQL is performed as a B-tree index search. Figure 5-44 shows the cost of this search given the order of the B-tree, which is the number of entries that can fit into a single database page. Given the PostgreSQL page size of 8kb and the fact that each entry consists of an oid value (8 bytes) and a pointer to a lower node (4 bytes), the order of a B-tree for oid values is approximately 666 nodes.

\[
B_T = \left\lfloor \log_{o+1}(R_T + 1) \right\rfloor
\]

where

- \( B_T \) = cost of B-tree search on table \( T \)
- \( o \) = order of B-tree
- \( R_T \) = number of rows in \( T \)

*Figure 5-44 - Cost function for a B-tree search.*

The costs for each operation are then calculated as follows:

**Object Selection**

The selection cost for an object is based upon the cost of the oid search to find the row relating to that object and the cost of selecting each of its fields. The field cost is weighted based on the proportion of access to the object that require the field value to be fetched to ensure that rarely needed fields do not contribute greatly to the cost of the object.
Object Update
Updating a class instance requires a separate UPDATE command to be issued to each table in which instances of that class may reside. Therefore, the cost of an update is the cost of an oid search on each table plus the cost of updating each field value weighted by the proportion of updates to the object that require the field value to be updated.

\[
C_A = S_A B_A + U_A N_A B_A \\
C_B = (S_B B_B + S_{B.a} B_A) + (U_B N_B B_B + U_{B.a} N_A B_A) \\
C_C = (S_C B_C + S_{C,a} B_A) + (U_C N_C B_C + U_{C,a} N_A B_A) \\
C = C_A + C_B + C_C
\]

where
- \( C_X \) = cost function for class X
- \( B_X \) = cost of B-tree oid search on table X
- \( N_X \) = number of tables containing instances of class X
- \( S_X \) = number of selects made to X
- \( U_X \) = number of updates made to X

Figure 5-45 - Cost function where fields are persisted using Foreign Key Aggregation.

Inlining a field eliminates the need for an oid search over that field, but increases the number of tables in which instances of the field’s type may appear, which affects the update cost of all other classes that contain a field of the same type. Therefore, the choice of whether a field should be inlined cannot be made using just information on the referring and referenced classes. Every time the persistence strategy for a path is changed, the cost function of the root class must be altered and the global cost function reevaluated.

\[
C_A = S_A B_A + U_A N_A B_A \\
C_B = S_B B_B + (U_B N_B B_B + U_{B.a} N_A B_A) \\
C_C = S_C B_C + (U_C N_C B_C + U_{C,a} N_A B_A) \\
C = C_A + C_B + C_C
\]

where
- \( C_X \) = cost function for class X
- \( B_X \) = cost of B-tree oid search on table X
- \( N_X \) = number of tables containing instances of class X
- \( S_X \) = number of selects made to X
- \( U_X \) = number of updates made to X

Figure 5-46 - Cost function where fields are persisted using inlining.

A search strategy was used to find the best mapping for minimising the global cost function by evaluating each possible combination of inlined paths. The search first calculates a list of paths for the object graph, excluding those involved in cycles. It then starts with the case where no paths are inlined, evaluates the cost function, then proceeds by inlining every combination of paths in the list. At each stage, a variable is updated to contain a reference to the minimum cost path found so far. The search terminates once all combinations of inlined paths have been evaluated. Since this is quite an inefficient algorithm, it is initiated by the user rather than run automatically.
Inheritance Hierarchies

Since persistence of inheritance hierarchies is implemented as an extension to the path-based inlining approach, the heuristics used to determine the cost of different methods of persisting references can also be applied to determining how inherited data should be stored. However, the costs of using different inheritance hierarchy techniques are subtly different from those for references. Inlined field references increase the execution time needed to update instances of the field type since all tables in which instances of that type may exist must be searched. This is not the case for inlined inherited fields, since at the point of update the runtime type of the object is known, so the tables involved can be calculated exactly.

The consequence of this is that any cost function for inheritance hierarchies that is based on the above techniques for determining execution time will tend to prefer inlining every field, producing a single table in which all instances of all types are stored. To prevent this situation from occurring, the cost function must take into account other factors that make a single supertable undesirable. Since this table would contain columns for all fields of all classes in the inheritance hierarchy, one such factor is unnecessary memory usage. Each persisted instance is only likely to have values for a small proportion of the columns in the table, so all remaining columns will be filled with NULL values. For large inheritance hierarchies containing many fields, this could lead to a large number of NULLs being stored for every persistent object.

Unfortunately, using memory-based metrics as part of the cost function is more difficult than using performance-based ones, since a level of acceptable memory waste must be set which will vary depending on the available disk space of the database server. Also, the actual waste caused by each NULL value is negligible, so this will only become a critical factor once the database has already grown to a substantial size.

One other potential factor is that a supertable will contain rows for every instance in the database, which will increase the time taken to search for particular instances. A cost measure based on this would need to track the number of instances of each class stored in the table and then start using the Table Per Class method to move instances to separate tables when a certain limit is reached.

Unfortunately, implementing this factor into the cost function of the persistence layer as it currently stands is unlikely to cause a rise in overall performance. This is because all queries produced by the persistence layer only search primary key values, for which B-tree indices exist. Lookups in these indices are very quick, even with a substantial number of rows in the table, so it is unlikely that the cost of searching for particular instances will ever become a critical factor in determining the cost of a mapping. The only benefit would be seen if the persistence layer were extended to build data prefetch paths (discussed in Section 9.2.1) that use searches over non-key fields to find only the particular instances required by each web page method before it is invoked.

5.4.3 Schema Change

If the heuristic optimiser terminates with a positive result, the next step is to alter the existing tables in the database such that object data are organised based on the improved mapping. To achieve this, the list of all types in the database is iterated over and the differences between the current inlined paths and the paths from the improved mapping are compared. If the current mapping for that type is different from the suggested mapping then the table must be altered to add any newly inlined paths and remove existing paths that are no longer needed.

Once the affected type tables have been found, the task is to execute a query that produces a single table containing all values for the inlined path tree of the new mapping for that type. The addSelectSql(..) method used during object reconstruction already takes as its parameter a tree
of paths and produces a single results table containing the values of all those paths inlined, so this method can be reused for applying the schema change. The only difference is that this time the initial `addSelectSql(..)` query is passed the new tree of inlined paths and a `SELECT INTO` command, which stores the results in a new table.

One complication is that the `SELECT INTO` command cannot be used to replace existing tables, therefore all data in the original table is temporarily duplicated and until the operation is complete, at which point the original table is dropped and the new table renamed to be the same as the old one.

### 5.5 Summary

This section described the implementation of an orthogonal persistence system for the Java language along with a number of optimisation techniques that were applied to improve the performance of accessing persisted objects. The task of providing orthogonal persistence was divided into a data storage phase and a data access phase, which were then further broken down into various data type groups, for which the design and implementation of custom persistence strategies were discussed.

One of the key requirements for the persistence layer was the ability to apply each of the following persistence techniques to different parts of the static class graph:

- Foreign Key Aggregation for persisting field values to separate tables.
- Single Table Aggregating for inlining fields into the referring class’ table.
- Table Per Class for persisting different subclass fields to different tables.
- Table Per Path for persisting each subclass to its own table.
- Table Per Inheritance Hierarchy for persisting all subclasses to the same table.

In the initial design, each of these techniques was considered separately, but through the course of the project the design evolved into a single algorithm of path-based inlining that would allow the above techniques to be applied to any part of the class graph.

Strategies for automatically optimising the table schema used to persist objects were also described and algorithms for achieving them presented. The use of generic type information was proposed for improving the performance of storing and accessing parameterised classes such as those in the Java Collections Framework. The design of a heuristic approach to evaluating the average performance of an object-relational mapping based on data usage statistics was also given and implementation details of using this information to alter the mapping for existing table schema were described.
6 The Presentation Layer

The role of the presentation layer is to provide an interface for developers to control the state of the web application and construct dynamic web pages to be displayed to users when they navigate to a particular URL. This section describes how these goals were achieved and the additional problems that arose during the design and implementation of this component.

6.1 Annotating Pages

One of the main requirements for the presentation layer is that it allows developers to easily add new pages to their web application. In the first design of this project, this was achieved by using a separate class for each page which was required to implement the IPage interface. This interface declared the handleRequest(...) method which would be invoked whenever a request for this page was received. This quite closely modelled the traditional approach of having a separate file for each page, and allowed the class name to be used as the URL pattern rule for executing that page.

```java
class MyPage implements IPage {
    public Document handleRequest() {
        return new Document("Hello, world!");
    }
}
```

**Figure 6-1 - Example page class implementing the IPage interface.**

Subsequent use of this design revealed that using a separate class for each page is not always desirable. If a group of pages were created for performing some piece of functionality then they would each have to be placed in a separate class, making the interactions between them less obvious. To improve upon this, the design was changed to use a method annotation, @Page, which would allow pages to be specified at the method level. This allowed pages that share the same persistent data to be written as methods in the same class, but sacrificed the ability to statically check links between pages.

```java
class MyPages {
    @Page public Document firstPage() {
        return new Document("Hello, world!");
    }

    @Page public Document secondPage() {
        return new Document("Goodbye, world!");
    }
}
```

**Figure 6-2 - Example page class using the @Page method annotation.**
6.2 Handling Web Data

Web application pages are request driven and controlled by the server, such that the server listens for web requests and then dispatches them to the relevant page method based on URL pattern rules. This means that communication between page classes is achieved through maintaining state, which can either be done at the user level (by placing data in the current session) or at the application level (by placing data in the database). Additionally, the user provides data to each web page in the form of either GET or POST data which is passed with the request. One of the requirements of the persistence layer is to provide a consistent interface for accessing and altering each of these different types of web data.

The initial design of this layer used the common approach to handling this data by providing the developer with two Maps as parameters to the page method which contained the session and request data. Persistent data was provided in the form of a third Map that returned PersistentCollection instances which would automatically persist any objects placed within it. This was quickly found to be a bad design since accesses to these Maps would be hidden throughout the page code, making it difficult to see which pages require which data. The use of a special Collection implementation for holding persistent data was also undesirable since it added clutter when persisting single instances and required persistent data and other web data to be retrieved through different interfaces.

To correct these problems, the design was altered to use field annotations to declare which fields of the page class correspond to the different forms of web data. The annotations were created; @Request, @Session and @Database. The original Map lookup was replaced by an automatic lookup based on the name of the field prior to any page methods in the class being executed. For database data, the use of the PersistentCollection class was eliminated so that objects of any type could be placed in a @Database-annotated field and would be automatically persisted.

```java
public MyPages {
    @Page public Document firstPage(Map<String, String> request, 
    Map<String, Object> session, 
    Map<String, PersistentCollection> database) {
        for (User user : (Collection<User>)database.get("users")) {
            if (user.getName().equals(request.get("user")) && 
                user.getPassword().equals(request.get("password"))) {
                session.put("user", user);
                return new Document("You are logged in");
            }
        }
        return new Document("Login failed");
    }
}
```

Figure 6-3 - Example code showing use of request, session and database data using the initial design.

To correct these problems, the design was altered to use field annotations to declare which fields of the page class correspond to the different forms of web data. The annotations were created; @Request, @Session and @Database. The original Map lookup was replaced by an automatic lookup based on the name of the field prior to any page methods in the class being executed. For database data, the use of the PersistentCollection class was eliminated so that objects of any type could be placed in a @Database-annotated field and would be automatically persisted.
**public** MyPages {

@Request String username;  
@Request String password;  
@Database List<User> users;  
@Session user;

@Page **public** Document firstPage() {
    for (User dbUser : users) {
        if (dbUser.getName().equals(username) &&
            dbUser.getPassword().equals(password)) {
            user = dbUser;  
            return new Document(“You are logged in”);
        }
    }
    return new Document(“Login failed”);
}

Figure 6-4 - Example code showing use of request, session and database data using field annotations.

This approach provides a much clearer and consistent interface than the previous design, but does have some issues. Firstly, since the web data is keyed by the name of the field, there may be conflicts where session data, request data and database data use the same keys. This is highlighted in Figure 6-4 by the need to rename the user request variable to username to avoid it clashing with the user session variable. There are also problems concerning dynamic data, such as forms using Javascript to return a variable number of request fields, since the names of these will not be known statically. This can be corrected by using array fields and a common naming convention to handle dynamic data, as shown in Figure 6-5.

```java
public MyPages {
    // Is filled with request data keyed by “words[0]”, “words[1]”, “words[2]” etc.
    @Request String[] words;

    @Page **public** Document firstPage() {
        return new Document(“You entered these words: “+MiscUtils.printArray(words));
    }
}

Figure 6-5 - Example code showing handling of dynamic data using an array request field.
```

### 6.3 Document Tree

One of the roles of the presentation layer is to allow developers to easily construct a representation of a page’s content which can then be decorated with the site design and displayed to the user. This is achieved by providing a set of classes for constructing a tree of document data and then using an iterative process to expand the tree into the correct format for display. Figure 6-6 shows the high-level design of the document tree classes. The presentation.document package provides a collection of high-level, format-independent classes for constructing the content of a page within a page class method,
while the presentation.xhtml package contains low-level classes that mirror the structure of an HTML document.

![Diagram showing the high-level relationships between the document tree classes.](image)

**Figure 6-6** - Diagram showing the high-level relationships between the document tree classes.

Once a page class method returns a Document object containing the content of the page to be displayed, a set of styles are applied that decorate the page with the site design. An IStyle interface is provided to allow new styles to be plugged into the presentation layer and invoked whenever a document is returned from a page method. An additional AbstractStyle class provides a set of methods that walk the tree and uses reflection to call methods on itself for each different tree node encountered. This allows subclasses of AbstractStyle to be written with separate methods for each node supported node type, which act like a set of rewrite rules for the tree.

![High-level design of style classes.](image)

**Figure 6-7** - High-level design of style classes.

When a style method for a particular node is invoked, the typical response is for it to use the data from that tree node and its children to create a new subgraph of lower-level nodes to replace it. However, they are not restricted to only altering just the node they are currently visiting; instead they can traverse parent and child references to access and modify any section of the tree. This is necessary for allowing the structure of the high-level document to be independent of the low-level document it will be transformed into. For instance, the high-level document may specify a set of styling hints for how particular text should be formatted. In HTML, these would be translated to tags inlined with the text for emboldening or italicising particular sections, whereas for PDF output the different text styles may need to be specified in the header of the document.
public class SimpleStyle extends AbstractStyle {
    public void processComponent(Document node) {
        node.replaceWith(
            new Html(
                new Head(
                    new Title(node.getTitle()),
                ),
                new Body(node.getChildren())
            )
        );
    }

    public void processComponent(Group node, Request request) {
        node.replaceWith(node.getChildren());
    }

    public void processComponent(Heading node, Request request) {
        node.replaceWith(new H1(node.getChildren()));
    }

    public void processComponent(Paragraph node, Request request) {
        node.replaceWith(new P(node.getChildren()));
    }
}

Figure 6-8 - Example style class for converting a Document to HTML.

One problem with this approach is determining the dependencies between each style. There are no guarantees that a style will replace a high-level node with a low-level node that is ready for display; instead, styles will often expand high-level nodes to other high-level nodes, so each style may need to be run multiple times. Currently, the tree styling is done in passes, where all the style classes involved traverse the document tree once in each pass, and is repeated until a pass occurs in which no alterations to the tree take place. This is quite an inefficient algorithm, since it requires the entire tree to be traversed multiple times. A possible improvement would be to mark sections of the tree that were altered in the current pass and only traverse those nodes in the next.

Once a Document has been transformed into the required structure, the tree must then be output in the format to be sent to the user. This task is performed by Formatter subclasses that traverse the tree and output a byte stream to be sent in the user response. For HTML and RSS, the XMLFormatter class can be used which walks the tree and outputs XML tags for each node. However, some formats like PDF do not use XML, in which case a custom Formatter implementation can be written and plugged in to produce the correct output.

6.4 Conclusions

This section described how the presentation layer was designed and implemented in order to provide the following features:

- An interface for adding pages to a web application.
- A consistent mechanism for accessing and updating request, session and database data.
Balancing simplicity and efficiency in Web applications

- A set of classes for constructing and styling format-independent trees of page content.

For each of these, an initial design was drawn up, implemented and tested. Its shortcomings were then highlighted and the design evolved to correct these issues. The end result is a system that is different from existing web presentation tools in that:

- Data used by page classes can be made persistent or stored in a session merely by annotating fields, requiring minimal impact on the design of the web application.

- Pages with similar data requirements or common functionality can be easily grouped together in the same class by using the `@Page` method annotation.

- The formatting and content of a page can be completely separated, allowing the same page methods to be used to generate documents in a number of different formats.

Enhancements to the existing system were then discussed, such as optimising the document traversal algorithm used when applying styles to a page to decrease the number of full-tree traversals that must take place.
7 The Server Layer

The server layer acts as a mediator between the user and the other layers of the application. It is responsible for intercepting and handling web requests as well as allowing developers to add, remove and configure their web applications. To achieve these tasks, the design was split into the following components:

The RequestDelegatingService component intercepts HTTP requests, applies a set of rules to the request URL to determine which web application should service the request and then executes the appropriate page class method.

The WebApplicationManager component contains the list of active web applications and is responsible for the dynamic loading of their classes.

The ConfigurationManager component constructs a user interface for adding, removing and configuring web applications.

The remainder of this section describes how each of these components was implemented, briefly evaluates their success and proposes potential improvements to their design.

7.1 RequestDelegatingService

This component intercepts HTTP requests and dispatches them to the appropriate web application based on a number of URL rules. This is achieved by wrapping an embedded server library called SimpleServer\(^\text{27}\) and registering hooks into it to listen for any HTTP requests to a particular port. When such a request is detected, a method in the RequestDelegatingService class is invoked and passed information about the request URL and any POST or GET data provided.

To find the relevant web application to delegate the request to, the request URL is parsed to extract the context path, file name and file extension. The following process then takes place:

- The list of running WebApplication instances is obtained from the WebApplicationManager.
- The list is iterated over and the context path of each is tested to see if it is a prefix of the context path of the URL.
- If a match is detected, then the page classes of that web application are iterated over to see if any classes and methods match the file name of the request URL.
- If no such classes are found then the file name is looked up in the list of resources declared by the web application.
- If no resources are found then the search through the list of WebApplication instances continues.

\(^{27}\) SimpleServer is an open source Java server available from http://simpleweb.sourceforge.net/.
Balancing simplicity and efficiency in Web applications

It should be noted from this algorithm that the ordering of the WebApplication instances matters since the context path of a web application must merely be a prefix of the URLs it can service. Therefore, each web application should be tested in descending order based on the length of its context path. By default, a special web application with no context path is set up to catch the case where no page class methods or resources were found that match the request URL. The sole purpose of this web application is to display error messages not caught by other web applications, such as the “404 Page Not Found” page.

Once a page class method is found for servicing a request, it is passed to the presentation layer which fills in the relevant request, session and database fields before the method is executed. After execution, the Document object representing the page content to be displayed is returned to the presentation layer for formatting. At this point, the server layer retrieves the necessary StyleManager for formatting the request through a lookup of the StyleManagers provided by the WebApplication instance based on the file extension of the request URL. This is then passed to the presentation layer which formats the page and returns a byte stream to be sent back to SimpleServer as the response to the request.

If a resource is found to match to a request URL then that resource is loaded from disk as a byte stream and passed in the response with the MIME type based on the file extension of the request. This is primarily used for serving static documents such as images or stylesheets that do not need to be built up dynamically by the web application. However, one problem with this implementation is that resources loaded this way are seen as dynamic documents by SimpleServer. This means that clients will disable caching of these files and re-request their contents for every page load.

To solve this problem, any static resources must be declared to SimpleServer separately to dynamic pages. This is less elegant because it requires each WebApplication instance to contact SimpleServer whenever a static resource is added, removed or refreshed. Another possibility is to alter SimpleServer to give the RequestDelegatingService complete control over the returned headers so that it can enable caching for URLs that map to static resources.

### 7.2 WebApplicationManager

When a web application is configured, the developer specifies a development directory, containing the page classes of the application, and optionally a resource directory, containing any static resources to be made available via URL requests. The WebApplicationManager is responsible for ensuring that the page classes and resources within these directories are loaded and made available to the RequestDelegateService when requested.

One of the requirements for the server layer was that it should be capable of detecting changes to web application page classes and automatically load the latest versions of altered classes. Unfortunately, there is no platform-independent way of hooking into the file system from within Java to be notified of file changes. Instead, a separate WebApplication instance is created for each web application configuration and runs a background thread to detect changes to the contents of the development directory every ten seconds. This process is only intended for use in development environments where the page classes may change regularly and can be disabled for use on production systems.

If a change is detected to a Java source file, the WebApplication automatically invokes the Java compiler via the Java Tools library to produce a temporary class file that can be loaded. Unfortunately, the following restrictions on Java class loading complicate this process:

- Running class files cannot simply be upgraded; Java Hot Code Replace allows for a limited set of changes to take place, but these are not extensive enough to make this a viable option.
• Different versions of the same class cannot exist within the same ClassLoader instance. Therefore, the new version of the class must be loaded in a separate ClassLoader.

• Classes of the same name loaded by different class loaders are type-incompatible. Therefore, attempting to pass an instance of the new class version to a class loaded with the old class’ ClassLoader will throw a ClassCastException.

To solve these problems, whenever any change to a web application class is detected, all classes from that web application are reloaded using a new ClassLoader. Any objects within the server that are holding references to instances of the old classes are then notified to drop their references such that the old classes may be garbage collected. One side effect of this is that any instances stored in the user session must be dropped, so all user sessions are invalidated whenever a page class is altered.

7.3 ConfigurationManager

Many of the web application frameworks discussed in Section 2 of this report make extensive use of XML files to configure different aspects of their web applications. This often leads to the same application being referenced from multiple configuration files, making updating the configuration difficult. This project attempts to improve upon this by providing a single, graphical interface in which all aspects of each web application can be configured.

A generic set of custom Swing components were created to simplify the task of creating configuration forms and wizards that can be easily integrated into the server’s GUI. These were then used to provide a set of forms to guide the user through adding web applications and altering their configurations. Using these forms, the following aspects of the server can be configured:

• The port on which the server intercepts HTTP requests can be changed (requires server restart).

• The development directory, resource directory and context path of a web application can be altered.

• The driver JAR file, host name, database name, username and password of a database connection can be altered.

Overall, the graphical interface provided by the server provides a single place in which all server settings can be viewed and altered without needing to worry about XML syntax. However, this interface will only work on machines that have a windowing system. For production servers, a separate command-line or web-based interface must be provided that gives access to the same functionality.

Figure 7-1 - Screenshot of the server GUI for viewing, adding and removing web applications.
7.4 Conclusions

This section described how the server layer of this project was designed and implemented to provide the following functionality:

- Interception and delegation of web requests to the appropriate web application based on the ordering of their context paths.
- Near-instant refresh of page classes through runtime recompilation and dynamic linking.
- A centralised configuration interface for viewing and altering all server settings.

During the implementation of these features, the following problems were found which require further work:

- When refreshing page classes, a background thread must be used to monitor changes to a web application’s development directory since file system notifications are not supported. This can introduce a delay before changes are reflected which will increase with the number of files to be monitored.

- Due to Java loading limitations, every change to a web application’s page classes require all classes for that web application to be reloaded. Care must be taken to ensure that all references to instances of the old classes are lost otherwise a substantial memory leak may occur. This could be improved by writing a custom `ClassLoader` which allows new and old versions of the same class to co-exist.

- Whenever a web application’s page classes are reloaded, all user sessions must be invalidated to ensure that references to old class instances are lost. This may be very inconvenient if a web application is changed regularly, since all user data will be lost and the developer may have to log in to his or her web application each time. To improve upon this, old class instances could be upgraded to new class instances using reflection if the class schemas are the same.

Overall, each of the above problems affect the efficiency of the server layer, but do not impact its required functionality as outlined in the Section 4 of this report.
8 Evaluation

In this section, the success of the project implementation in fulfilling the goals set out in Section 4 is assessed. For the persistence layer, a quantitative analysis is performed on the performance of storing and retrieving objects using the various optimisation techniques described. Shortcomings of the existing implementation are then highlighted with a view to future work. For the presentation and server layers, a more qualitative analysis is done on the usability of the system and whether it meets the minimum requirements set out in the specification.

8.1 Persistence Layer

This section provides a quantitative evaluation of the performance of the persistence layer implemented for this project. The correctness of the persistence mechanism is assessed based on the results of unit tests designed to verify the fulfilment of its functional requirements. The performance requirements are then evaluated by assessing the overhead of using the automatic persistence mechanism, comparing the advantages and disadvantages of the lazy loading technique and testing the benefits of using the heuristic optimiser to improve the object-relational mapping.

8.1.1 Provision Of Orthogonal Persistence

One of the core goals of the persistence layer was the provision of an orthogonal persistence mechanism for the Java language. In Section 4, this was refined into a set of requirements that specified the minimum functionality to be provided by this mechanism. In order to test whether these requirements had been fulfilled and to give a progress indicator during the course of this project, the implementation of the persistence system followed a test-first design strategy. Before any new functionality was added, a series of unit tests or integration tests were written that would assess whether the implementation fulfilled its purpose. The TestNG framework was used to allow these tests to be easily run whenever the codebase was altered to detect any unexpected side affects caused by the modifications.

The tasks performed by the persistence layer were broken down into those responsible for the creation of tables, construction of INSERT and UPDATE commands, construction of SELECT queries and the reconstruction of objects. For each of these categories, tests were then written for each of the supported data types outlined in the specification. Once all the tests in each of these categories passed for a particular data type, an integration test was written to test the complete round trip of storing and retrieving an instance of that type in order to ensure that the components in the persistence layer would interact with one another in the required manner.

Overall, 190 tests were written during the progress of this project. The following subsections describe examples of these tests for each of the main aspects of the persistence mechanism.

---

28 TestNG is a Java unit test framework similar to JUnit. It can be obtained for free from http://testng.org/.
8.1.1 Table Creation (37 Tests)

Each of the tests for table creation take a particular Java type, an array of inlined paths and a description of the expected table schema for that type as parameters. The `addCreateSql(..)` method for the given type is then invoked and its resulting `CREATE TABLE` command compared against the expected schema. Figure 8-1 shows an example whereby a type containing primitive, Collection and Image fields is tested.

```java
@Test public void createObjectWithInlinedPaths2() throws PersistenceException, SecurityException, NoSuchFieldException {
    // Set primitives to be immutable for this test
    PrimitiveSqlProvider.immutable = true;
    createObjectWithInlinedPaths{
        new Pair[] {
            new Pair(B.class, new Pair[] {
                new Pair("root_a", Long.class),
                new Pair("root_a_intField", Long.class),
                new Pair("root_a_intField_value", int.class),
                new Pair("root_collection", Long.class),
                new Pair("root_image", Long.class)
            })
        },
        new Pair[] {
            new Pair(B.class,
            new Path[] {
                new FieldPathEntry(B.class.getDeclaredField("a")),
                new FieldPathEntry(A.class.getDeclaredField("intField")),
                new PrimitiveValuePathEntry(int.class)
            })
        });
    }
}
```

**Figure 8-1** Table creation test example.

8.1.2 Row Insertion (30 Tests)

The tests for row insertion take an object and a description of the expected row data as parameters, then execute the `addInsertSql(..)` method corresponding to the object’s runtime type. The resulting `INSERT` statement is then compared against the expected row contents. Figure 8-2 shows an example in which the insertion of a primitive array is tested.

```java
@Test public void insertArray() throws PersistenceException, SecurityException, NoSuchFieldException {
    // Set primitives to be mutable for this test
    PrimitiveSqlProvider.immutable = true;
    int[] ints = new int[] { 7, 9, 2, 5, 1, 2 };
    insertObject(ints,
        new Pair[] {
            new Pair(int[].class,
            new Path[] {
                new Pair(SqlProvider.OID_COLUMN_NAME, 1),
                new Pair(SqlProvider.RUNTIME_CLASS_COLUMN_NAME, "[I"),
                new Pair("root_index", 0),
            })
        });
}
```

**Figure 8-2** Row insertion test example.
8.1.1.3 Row Selection (22 Tests)

These tests take as parameters the static type and oid of the object, an array of inlined paths and a description of the format of the result table. The SqlProducer for the given static type is found and the result of its addSelectSql(..) method compared with the given result table format. Figure 8-3 shows an example where an image is being fetched.

```java
@Test public void selectImage() throws PersistenceException {

    PrimitiveSqlProvider.immutable = true;
    // SELECT root.___cuxca_oid AS root, root.root_data AS root_data
    // FROM ToolkitImage AS root
    // WHERE root.___cuxca_oid = 1
    selectObject(ToolkitImage.class, 1,
        new FieldIdentifier[] {
            new FieldIdentifier("root", SqlProvider.OID_COLUMN_NAME, "root"),
            new FieldIdentifier("root", "root_data", "root_data")
        },
        new Pair[] {
            new Pair("ToolkitImage", "root")
        },
        new Pair(new FieldIdentifier("root", SqlProvider.OID_COLUMN_NAME), 1)
    );
}
```

Figure 8-3 Row selection test example.

8.1.1.4 Object Reconstruction (14 Tests)

Object reconstruction is tested by providing the static type of the object to be reconstructed and the contents of any result tables containing the required object data. The SqlProducer for the given static type is found and its reconstructObject(..) method invoked. The reconstructed object is then obtained and its data values inspected against those provided in the result table. Figure 8-4 shows an example where an object with primitive fields is reconstructed.
@Test public void reconstructObject2() {
    E e1 = new E();
    e1.a = new A();
    e1.a.intField = 5;
    e1.a.floatField = 32.1f;
    e1.a.shortField = 2;
    e1.a.doubleField = 12321.4;
    e1.a.byteField = 12;
    e1.a.booleanField = false;
    e1.a.charField = 'h';
    e1.a.longField = 3423424L;
    e1.a.stringField = "Goodbye, world!";

    SqlProducer p = reconstructObject(E.class,
            new Pair[][] {
                new Pair[] {
                    new Pair("root", 1L),
                    new Pair("root_a", 2L),
                    new Pair("root_a_intField_value", e1.a.intField),
                    new Pair("root_a_floatField_value", e1.a.floatField),
                    new Pair("root_a_shortField_value", e1.a.shortField),
                    new Pair("root_a_doubleField_value", e1.a.doubleField),
                    new Pair("root_a_byteField_value", e1.a.byteField),
                    new Pair("root_a_booleanField_value", e1.a.booleanField),
                    new Pair("root_a_charField_value", e1.a.charField),
                    new Pair("root_a_longField_value", e1.a.longField),
                    new Pair("root_a_stringField_value", e1.a.stringField)
                }
            });

    E e2 = (E)p.getObject();
    assert e2.a.intField == e1.a.intField;
    assert e2.a.floatField == e1.a.floatField;
    assert e2.a.shortField == e1.a.shortField;
    assert e2.a.doubleField == e1.a.doubleField;
    assert e2.a.byteField == e1.a.byteField;
    assert e2.a.booleanField == e1.a.booleanField;
    assert e2.a.charField == e1.a.charField;
    assert e2.a.longField == e1.a.longField;
    assert e2.a.stringField.equals(e1.a.stringField);
    assert p.getOid() == 1L;
}

Figure 8-4 Object reconstruction test example.

8.1.1.5 Round Trip (55 Tests)

Each of these tests constructs an object and passes it to the persistence layer to be persisted. The object’s oid value is then used to retrieve it from the database, and the retrieved and persisted instances compared for data equality. The ObjectCache is flushed in between the object being persisted and retrieved in order to force retrieval from the database rather than main memory. Figure 8-5 shows an example where persistence of a HashSet containing some null values is tested.

@Test public void storeNullHashSet1() throws PersistenceException {
    HashSet<Integer> collection1 = new HashSet<Integer>();
    for (int i = 0; i < 10; ++i) {
        if (i % 3 == 0) {
            collection1.add(null);
        }
    }
}
8.1.2 Overhead Of Automatic Persistence

In the implementation section of this report, a number of complex algorithms were described for automatically generating SQL for storing and retrieving objects from a relational database. However, each of these algorithms adds an additional runtime overhead compared to the use of manually written SQL queries. This section tests this overhead by comparing the proportion of time taken to store and retrieve objects that is spent in the automatic persistence code against the time taken for the generated queries to be run on the database.

8.1.2.1 Primitive Values

In this test, the time taken to construct the SQL for retrieving different primitive value types was recorded along with the execution time of the queries on the database. For each type, the times were averaged over 100 runs.
Despite the fact that primitive values are very quick to retrieve from the database, the overhead of constructing the SQL for accessing them and reconstructing them remains below 10% of the total execution time.

8.1.2.2 Object Graphs

In this test, a number of object graphs containing varying numbers of objects (field references) were persisted. The time taken to retrieve the object graph data was then measured and compared against the time taken to construct the query and reconstruct the object graph. For each graph size, the test was run 100 times and the results averaged.

![Figure 8-7 - Graph of execution time breakdown for retrieval of object graphs.](image)

The results shown in Figure 8-7 indicate that, although reconstruction of object graphs require separate SqlProducers to be created for each object being reconstructed, the overhead of doing so is less than 5% of the time taken to execute the generated query.

8.1.2.3 Arrays

This test assessed the time taken to fetch and reconstruct nested arrays of different sizes compared to the time taken to execute the SELECT statement to retrieve the array data. For each array size, the test was run 100 times and the results averaged.

The results of this test given in Figure 8-8 show that, despite the complex algorithm used to distribute array data, the overhead of automatic persistence still remains less than 10% of the total execution time for fetching the array. These results also suggest that the overhead decreases as the size of the array increases, suggesting that the automatic persistence system should scale well to larger problems.
8.1.2.4 Conclusions

Despite the use of complex algorithms for constructing queries and distributing data amongst the reconstructed objects, the overhead of using the persistence layer was consistently less than 10% of the total time taken to execute the query and return the requested objects. This shows that there is an acceptable trade-off the use of automatic query construction over manually coding the queries in the web application itself. Further optimisations may also be possible to reduce this overhead, such as caching of commonly used queries to eliminate a full traversal of the static class graph for frequently accessed classes.

8.1.3 Lazy/Eager Loading Comparison

The lazy loading technique delays the retrieval of an object’s field values until they are required by the application, whereas eager loading will retrieve all field values of an object when it is requested. Although the former avoids retrieval of unnecessary data, it does so at the expense of executing multiple queries in separate communications to the database. Since each of these queries is typically fine grained, fetching only a single field at a time, there is little opportunity for query optimisation. To test this, various scenarios were constructed and their performance under both lazy and eager loading was recorded and compared.

8.1.3.1 Test One

The first test involved fetching only a primitive field of a class that also contained an array. The size of the array was varied and the average execution time for both lazy loading and eager loading over 100 runs was compared. The classes used in this test are shown in Figure 8-9. To begin with, none of the fields of the B class were inlined, so each requires a JOIN between the B table and the field value table.

The results in Figure 8-10 show that lazy loading is advantageous is cases where only a small proportion of an object’s data is required. Since both the bi and as fields of the B class are persisted using Foreign Key Aggregation, retrieving them with eager loading requires an expensive JOIN operation, which lessens the relative cost of executing multiple SELECTs with the lazy loading approach. As expected, as the amount of unused data increased, the execution time for eager loading increased while the execution time for lazy loading remained constant.
class A {
    int ai;
}

class B {
    private int bi;
    private A[] as;
    ... getters and setters ...
}

Figure 8-9 - Class layout for objects used in test one.

![Comparison of eager and lazy loading of B objects](image)

Figure 8-10 - Test one results: comparison of eager and lazy loading of B objects.

The test was then re-run with all fields of the B class inlined. This reduces the cost of eager loading by eliminating the need to JOIN tables to fetch the contents of the array. These results, shown in Figure 8-11, demonstrate the additional overhead of executing multiple queries in lazy loading. Despite the fact that eager loading is constructing the bs array, it is faster than lazy loading for array sizes less than 25. However, since the difference between lazy loading and eager loading is negligible for small array sizes but substantial for larger ones, lazy loading is still the preferred technique for this situation.

![Comparison of eager and lazy loading of B objects with inlined paths](image)

Figure 8-11 - Comparison of eager and lazy loading of B objects with inlined paths.
8.1.3.2 Test Two

In this test, a List was constructed, filled with 1000 elements and then a varying proportion of those elements were accessed. Each element in the List was an instance of a class with a single primitive field and was not inlined into the List table. For each number of elements accessed, the test was run 100 times and the results averaged.

![Comparison of eager and lazy loading for Lists](image)

**Figure 8-12** - Comparison of eager and lazy loading for Lists.

The results in Figure 8-12 show that the cost of executing separate SQL queries for each List element greatly outweighs the cost of fetching the entire List in a single query. This is likely to be due to the optimisations that take place in the single query when performing the JOIN between the List and element tables, as well as the decreased overhead of communicating with the database only once.

8.1.3.3 Test Three

This test was designed to assess the performance hit suffered by non-persistent List instances as a result of the lazy-loading enhancement phase. In this test, a non-persistent ArrayList instance was filed with 1000 objects, then the number of elements accessed was varied and the time to perform the accesses recorded. The ArrayList class was then enhanced and the test re-run.

The results in Figure 8-13 show a substantial performance hit for accessing elements from an ArrayList after it has been enhanced, despite the fact that the ArrayList being accessed is not persistent and therefore does not need to access any data from the database. This is due to the fact that both persistent and non-persistent instances may co-exist in an application, and since enhancement alters the class rather than individual instances, a check must be performed on every element access to determine whether the particular instance involved is persistent or not. In this case, the performance hit suffered is the cost of looking up the ArrayList instance in the ObjectCache, which is a HashMap lookup, for every element access.
8.1.3.4 Conclusions

The lazy loading technique has been shown to offer substantial performance benefits in cases where the amount of unnecessary data loaded through eager loading is high. However, this evaluation has shown that there are also many cases where lazy loading provides a large performance decrease due to executing separate, fine-grained SQL statements for each piece of required data. Further work must be done to detect these cases and aggregate the multiple SQL queries together into a single, coarse-grained query for selecting only the required data. Strategies for achieving this are discussed in Section 9.2.

These tests also show that the Hot Code Replace approach used to implement the lazy loading technique has the drawback of affecting the performance of non-persistent instances of an enhanced class. The cost of checking whether an instance is persistent can add significant overhead to object getter and setters, which could cause unacceptable performance for commonly-used classes. This could be improved by speeding up the persistence check, which is discussed in Section 9.

8.1.4 Schema Optimisation Heuristics

The heuristic optimiser gathers field access statistics for each object and uses a set of heuristics to decide when paths should be inlined. To assess the quality of these heuristics, a set of classes were created and a predefined set of usage statistics given to the heuristic optimiser. The results of accessing instances of the classes were then gathered and compared to the default strategy of using Foreign Key Aggregation.

8.1.4.1 Test One

In this test, three classes were created and assigned different access and update statistics. The heuristic optimiser was then applied to find the inlining strategy that minimises the overall cost function. Each of the possible inlining strategies were then tested by creating instances of each class, then persisting and updating them in the manner dictated by the usage statistics. The execution time of each operation was recorded and averaged over 100 runs then compared to see whether it matched the estimate produced by the heuristic optimiser.
class A { int a; }
class B { A a; }
class C { A a; }

Usage statistics:  \( R_A = 20, S_A = 50, U_A = 5 \)
\( R_B = 50, S_B = 50, U_B = 10, S_{B.a} = 50, U_{B.a} = 5 \)
\( R_C = 10, S_C = 5, U_C = 20, S_{C.a} = 0, U_{C.a} = 10 \)

where  \( R_X \) = number of rows in X table  
\( S_X \) = number of selections made to X  
\( U_X \) = number of updates made to X

Heuristic cost results:

- Cost with no inlining:  205
- Cost with B.a inlined:  170
- Cost with C.a inlined:  220
- Cost with B.a and C.a inlined:  195

---

**Figure 8-14** - Comparison of overall execution time for different field persistence strategies.

The results in Figure 8-14 show that the strategy which reduces the overall execution time is also the strategy which scored the lowest in the cost heuristics. It also shows that a hybrid strategy can be advantageous even in simple cases such as the one above, where only the B.a path benefits from inlining while inlining the C.a path would cause a decrease in performance. This demonstrates that the cost function successfully distinguishes between paths with a high select rate and those with a high update rate.
8.2 The Presentation Layer

The presentation layer of this project provides a simple, concise interface for managing the persistence of objects and constructing web pages. Web developers can make use of a consistent interface for handling request, session and database data with minimal impact on the design of the application. This is different from many of the web application frameworks discussed in Section 2, which typically delegate the task of handling different types of web data to different tools, each with customised methods for accessing and updating persistent objects.

The use of Java annotations to denote the persistence of fields provides a number of advantages over other methods. Each web page class now declares its persistent state with its field declarations rather than in calls to persistence systems hidden in method bodies, so it is easier to see how each page class shares data with the other pages in the web application. This system also allows for greater optimisation opportunities since the persistent fields of a class are known statically, so their usage can be traced through the code using data flow analysis. This can then be used to prefetch data before the page method is executed, allowing multiple queries to be aggregated together.

For construction of web documents, a simple document model is provided by the persistence layer that allows page content to be built up independently of the final format in which it is displayed to the user. Style visitors can then traverse the document and transform it into a lower-level, format-dependent structure. Unlike HTML stylesheets, these Style classes have complete access to create, delete and modify any node in the tree, which is important for allowing the same document to be transformed into various heterogeneous formats such as RSS and PDF. Although existing tools such as SiteMesh (see Section 2.4.4) attempt to offer similar functionality, they are restricted to altering documents that are already in the HTML format, which limits their flexibility. Finally, this system allows developers to easily extend the existing styles and components to create their own. This allows any groups of components that are common to a large number of pages to be combined into a single tree node, greatly reducing the amount of code repetition in page class methods.

However, during the implementation of the presentation layer, it became apparent that using Java to construct low-level document trees, such as pages in pure HTML, is very time consuming. This is often necessary when constructing Style classes for decorating a content document with the graphical design of the site, which is often the responsibility of specialised graphics artists who may not be familiar with the Java language. Raw HTML code could be embedded into the document tree as a text node, however this is undesirable since other Style classes would then not be able to access and alter its structure if necessary, losing a large amount of flexibility.

8.3 The Server Layer

The main requirements of the server layer specified in Section 4.3 of this report were to:

- dynamically compile and load web applications.
- intercept HTTP requests and forward them to the correct web application.
- provide a user interface for configuring the application.

To achieve dynamic loading of web applications, a background thread is created to monitor the development directory of the web application in order to detect changes to the class or source files. This fulfils the minimum requirement of being able to load changes into the server at runtime, however the strategy of polling the file system is quite inefficient and can lead to delays between a change being
made and it being reflected in the web application. Further work needs to be done to improve the scalability of this approach.

Interception of web requests is managed by wrapping an existing server library, SimpleServer, and then using string matching techniques on the request URL to find the correct web application to forward to. The use of nested context paths provides a clean delegation mechanism between web applications and the inclusion of a resources directory allows static files to be easily linked in with dynamic content. However, the SimpleServer interface currently does not allow these resources to be cached on the client end, resulting in the overhead of re-loading static files each time they are linked. Minor alterations to the SimpleServer API should be able to correct this shortcoming.

The server layer also provides a user interface containing forms and wizards to guide the developer through the process of configuring web applications and database connections. This is an improvement over previous XML-based configuration schemes, since it provides instant, descriptive feedback if the entered values cause the server to error. The disadvantage of using a graphical interface for configuration is that it will not run on machines that do not have a windowing interface. An additional command-line or web-based configuration system should be provided for these cases.

8.4 Summary

This section compared the implementation of each of the three main components of this project against the goals set out in Section 4. For the persistence layer, a number of tests were performed which demonstrated minimal overhead in using an automatic persistence system and a performance increase for optimised queries in some cases. However, certain situations that were not apparent from the specification may cause a performance decrease. Lazy loading was found to be beneficial when the proportion of accessed class fields was low, but the cost of executing multiple queries for each data value makes it inefficient for traversing lists. The overhead added to method calls for non-persistent instances after a class is enhanced to take advantage of lazy loading was also found to be high, so a different technique that can advise only persistent instances should be researched before this can be used as a viable optimisation for commonly used classes.

For the presentation layer, the use of field annotations to denote the persistence of field values was found to simplify the interface for accessing and updating web data and allow the persistence requirements of a web application to be altered with minimal changes to its design. The mechanism for building dynamic page content and decorating it in a separate pass with format-specific styles was found to aid reuse of common page components and allow page content generated by the same page method to be viewed in multiple different formats. However, since there is no way of importing static data into the document tree, creating styles for decorating content with a complex graphical design can be a very time-consuming task. This could be improved by providing an automatic mechanism for parsing static content into the document tree format. The server layer was also assessed and found to have some minor efficiency shortcomings, but overall managed to meet all of the criteria required by the specification of this project.
9 Conclusions

This project was motivated by the fact that existing products for constructing dynamic web pages are either too complex, such that rapid development of new functionality is difficult, or too inefficient, making them unviable for large-scale systems. A number of existing tools and technologies associated with web application development were tested and compared to produce a set of goals that, if achieved, would improve the current state-of-the-art in this area. These goals were then refined into a large set of requirements which were used to provide a specification for the project implementation.

During the design and implementation of this project, a number of unforeseen problems were identified which added an extra level of complexity to the final product. During the evaluation of this project, the effects of these problems on the performance and usability of the system were determined. The following conclusions were drawn:

- The overhead of generating efficient SQL by the automated mapping layer provided by this project is typically less than 10% of the total execution time spent executing the query and reconstructing the requested objects.

- The lazy loading optimisation implemented in this project provides a significant performance increase in cases where only a small proportion of an object’s data is required by the web application. However, when a large number of fields are traversed, lazy loading can be significantly slower due to the inability to optimise between the SELECT queries executed to fetch each field.

- Adjusting the persistence technique used for storing references between objects automatically based on a heuristic cost function can provide an overall increase in performance for particular patterns of object usage. However, finding a suitable heuristic for automatically adjusting the persistence technique for class hierarchies is complicated since the advantages and disadvantages of inlining fields are not just based on the access and update patterns but also the types of searches performed over each field.

- The provision of a presentation layer that clearly separates content from style makes the task of writing new web pages easier through the reuse of existing page components and the ability to delegate the task of formatting the response elsewhere. However, for complex graphical designs, the ability to construct low-level, format-specific content is required. Therefore, a compromise must be found between hiding the details of the resulting format and providing easy access to low-level functions where appropriate.

Many of the problems faced during the implementation of this project were due to the constraints of the Java language and SQL interface. The following subsections outline two possible improvements in these areas that would allow the design of this project to be significantly simplified.

Low-Level Database Access

Many of the algorithms implemented in the project were necessary to construct efficient SQL queries for storing and retrieving the necessary data. However, the SQL and relational model itself has a number of shortcomings that decrease the efficiency of the persistence layer. For complex queries that JOIN multiple tables, a query planner component running on the database will typically perform a heuristic search of possible query transformations in order to find the best strategy for accessing the database. However, in many cases the persistence layer knows enough information about how data will be accessed to discount a number of strategies from this search. Unfortunately, the SQL standard is not expressive enough to allow this information to be passed to the database.
Low-level access to the functions of the database would allow this information to be utilised and would also remove much of the overhead of constructing a query in the SQL language which must then be parsed by the database. It would also allow for greater control over the format of the results returned by the database, since often queries over persisted arrays result in a lot of duplicated data being returned that is then discarded by the array reconstruction algorithm. Unfortunately, it would be difficult to supply these features without losing the database independence of the SQL language.

Improved Java Tool Interface

The Java Virtual Machine Tool Interface currently provides a number of features that allow the state of the Virtual Machine to be inspected and altered by external processes. This project already makes use of its Hot Code Replace functionality in order to enhance classes to load their persistent data lazily, but due to the restrictions on allowed alterations to a class this technique can cause a performance decrease for non-persistent instances of those classes. This is mainly due to the need to perform a HashMap lookup at the start of each enhanced method in order to check whether the current instance is persistent.

If the features of Hot Code Replace were extended to allow further runtime changes to a class then the performance hit of lazy loading could be significantly reduced. By allowing new fields to be added to a class, a simple boolean flag could be included in each instance to identify whether it is persistent. This would reduce the cost of checking for persistence to the cost of a field access.

However, the need to use Hot Code Replace for lazy loading could be completely eliminated if functionality for being notified of field accesses within the Java Virtual Machine was provided. This would allow all data access to persisted classes to be intercepted, including non-private fields which cannot be detecting with the current technique.

9.1 Contributions Of This Project

This project started out by analysing a number of existing tools and technologies used to construct web applications and outlined a number of areas in which the current state-of-the-art could be improved. It soon became apparent that persistence of object data was the main source of complexity and inefficiency of most systems. Whereas most frameworks provide reasonably low-level access to the database to allow developers to define custom object-relational mappings, this project proposed a system that would automate all interaction with the database and allow persistent objects and normal Java objects to co-exist indistinguishably.

A similar persistence system for the Java language had been previously proposed by a team at Glasgow University back in 1996 (see Section 2.3.1). That project defined three key design rules that the specification of this project is based upon, namely orthogonal persistence, transitive persistence and persistence independence, which together would provide a seamless system for storage and retrieval of Java objects. However, their implementation was based upon alterations to the Java Virtual Machine, which was quickly discounted as a viable solution for an industrial application.

Since the PJava project, a number of technologies have appeared, such as Aspect-Oriented Programming and Java Hot Code Replace, which this project evaluated for providing an orthogonal persistence system without JVM modification. Overall, a working implementation was produced that showed how a persistence system that preserves the semantics of the Java language could now be achieved on the standard Sun JVM.

An area that was neglected in the PJava project, but is pivotal to the goals of this one, is the efficiency of the storage mechanism used. This was provided by two approaches; first the dynamic alteration of the object-relational mapping and secondly through lazy loading. Five common methods of persisting
Object hierarchies were researched and generalised to a single algorithm of path inlining for constructing hybrid object-relational mappings. A feedback mechanism was then introduced which gathered statistics on the way objects are accessed and adjusts the mapping in order to minimise the overall execution time.

The project evaluation showed that the strategies for improving efficiency used in this project can provide a performance increase in some situations, although further work has to be done before complex data usage patterns can be detected and handled efficiently. However, this project has provided an extensible system on top of which these optimisations can be easily constructed given more time.

Part of this project was also devoted to providing a suitable interface for building web pages. A system of field annotations was used to provide a unified interface to request, session and database data which was not provided by any of the existing web application frameworks studied. This allows web application developers to alter the lifetimes of their objects with minimal impact on the application code.

A system for constructing format-independent pages and transforming them with styles was also implemented to allow the same web pages to be displayed in multiple formats. Existing systems such as CSS, SiteMesh and XSLT have attempted to provide similar functionality, but were either tied to HTML or not easily manipulated from within Java. The solution provided by this project is more intuitive for experienced Java programmers, but can be too verbose when constructing the low-level page design. Further work must be done to find a compromise in this area.

### 9.2 Future Work

During the implementation of this project, many new problems arose that couldn’t be addressed in the time allotted. This section covers some of these problems and describes how this project could be extended to solve them.

#### 9.2.1 Prefetch Paths

The lazy loading technique was implemented in this project to avoid fetching unnecessary data from the database at the expense of executing separate SQL statements to retrieve the data when required. Unfortunately, there are some situations in which this can provide very poor performance, particularly involving traversal of Lists. One way to improve upon this is to analyse the usage of persistent objects within the page class methods to determine which fields or elements are going to be accessed prior to the page being executed and fetch them in a single statement. This could be implemented in two stages:

- **The page class could be analysed to find the fields marked with the `@Database` annotation, then simple data flow analysis could be performed to trace field accesses and construct a set of prefetch paths for the object. By reusing the path tree implementation used for inlined paths in the persistence layer, these prefetch paths could be integrated into the persistence mechanism with minimal changes to the existing code.**

- **Further analysis of the page classes could detect conditionals and loops in which persistent objects are used. For each field access, a set of conditional expressions that must be true at that point could be constructed and used to annotate the prefetch path. If these conditionals only contain constants, known data values (i.e. request or session data) or persistent values then they can be added to the `WHERE` clause of the SQL query generated for fetching that data.**

The main goal of constructing prefetch paths would be to translate the Java code that utilises the persistent objects to SQL code as far as possible. A further step would be to remove Java code entirely
if the logic it performs is fully catered for by the SQL. However, a number of cases must be carefully considered when implementing such a system:

- Virtual method calls may dispatch to a variety of different methods depending on the runtime type of the target object. Any data flow analysis should take care when tracing into these calls to ensure that all possible target types are considered.

- Any changes made to a class will apply to both persistent and non-persistent instances of that class, so checks must be performed to ensure that the semantics of the affected methods are not altered.

9.2.2 Heuristic Cost Function Improvements

Currently, the heuristic optimiser uses a cost function to determine when referenced values should be inlined into their referring table. In Section 5.4.2, a number of extensions to this cost function were suggested to distinguish between the scenarios in which different inheritance mappings would be beneficial. Unfortunately, these were found not to be of any benefit to the current implementation of the persistence layer, since they relied on non-index searching to have a large influence on the access times for persistent objects.

One of the optimisations discussed in the proposal of prefetch paths is the ability to propagate conditional statements through the web application code to determine which properties must be true when field accesses to persistent objects occur. If these conditionals are then added to the SELECT queries executed on the database then searches over non-index columns may become a regular occurrence. In this case, the cost function could be adapted to decide whether different inheritance persistence strategies should be used or to determine whether an overall performance gain will be achieved by creating indices on the most commonly-searched columns.

Further work could be done to research other factors that distinguish between situations where different inheritance hierarchy mappings would be appropriate. So far it has been assumed that the cost of updating instances would not be affected by the size of the table in which the instance is stored, however this may not be the case if many indices are declared over the columns of that table, since each of these must be updated accordingly. To determine whether this is a worthwhile factor to include in the cost function, a series of tests could be run to compare the performance of various database operations on large tables both with and without indices.

9.2.3 HTML Import

The main problem with the current implementation of the presentation layer is that constructing low-level document trees is very time-consuming and may need to be performed by dedicated graphics staff who have no knowledge of Java. This could be improved by allowing XML files to be loaded into the presentation layer for use in the style transforms. Since other styles must be able to alter the structure of the loaded XML, it must be parsed and converted into the presentation layer’s document format before being used. Further work could also be done to allow Velocity templates or similar to be loaded and run, allowing for primitive style transforms to be performed in an XML-like environment.
9.3 Summary

Overall, this project fulfilled many of the goals it set out to achieve by providing a tool with which web developers can easily construct dynamic web sites. Various optimisation techniques were implemented and tested to fulfil the efficiency goals that motivated this work, some of which were found to be successful at increasing performance and some of which were not. For those areas that didn’t perform as expected, tests were run to determine the likely situations under which they caused a decrease in performance and possible improvements were suggested as future work.
10 Bibliography

The following are references to publications used to research this project and may provide useful information to those interested in this subject.


[Codd 70] E Codd. ACM Computing Surveys. A Relational Model of Data for Large Shared Data Banks.


[Field and Hutton 96] A J Field and J A R Hutton. The Integration of Functional Languages and Relational Databases.


Balancing simplicity and efficiency in Web applications


[Yeganeh 04] Z Yeganeh. Nestor: A Tool for Advising on & Applying Domain Specific Optimisations to JDBC.
11 Appendix

This section elaborates on some of the details that were only summarised in the main report due to space and consistency reasons.

11.1 $M^2ORM^2$ Representation

The $M^2ORM^2$ and $M^2ORM^2+HIE$ notations are a visual representation of a mapping between the structure of a class or set of classes and a relation or set of relations. This representation is used throughout this report as a concise way to describe various mapping strategies. This section is only meant to serve as a brief summary of the notation for the purposes of reading this report. For a full explanation, please consult the original papers [Cabibbo and Porcelli 03] and [Cabibbo and Carosi 05].

The $M^2ORM^2$ (Meet-in-the-Middle Object/Relational Mapping Model) representation used is a graph where each node contains a mapping between either a set of classes (a $c$-cluster) and a single relation or a set of relations (an $r$-cluster) and a single class. Classes are represented by boxes with white headings and camel-cased names, while relations are denoted by grey headings and lower-cased names. Arcs between nodes represent the relationships between the objects they comprise; for instance, if an object contains a reference to another object, then an arc will exist between the mapping node for the referer object and the mapping node for the referee object. Similarly, for relations, an arc will exist between a foreign key field and the table to which the key belongs.

![Diagram of $M^2ORM^2$ representation]

**Figure 11-1 Example of relationships between mapped objects in $M^2ORM^2$ representation**

In the example in Figure 11-1, the StarterMotor class contains a reference to the Solenoid class, denoted by the directed arc between the two classes. This is reflected in the relational mapping, where the solenoid attribute of the startermotor relation is a foreign key into the solenoid relation, denoted by the directed arc between the two relations. There is also an arc between the two nodes to show that such a relationship exists between some of their components.

Mappings between the structure of classes and relations within nodes are represented by dotted lines between members and attributes. Attributes that form natural keys are represented by the \{NK\} constraint after their names, while artificial keys (those generated by the database) are represented by the \{AK\} constraint. Attributes which are foreign keys into other relations are denoted by \{FK\} and attributes which may have null values are denoted by \{null\}. In the original papers, member variables of classes were not explicitly typed (they were all assumed to have String type), but this has been expanded for the purposes of this report by stating the type of each variable.
The M²ORM²+HIE representation extends the original representation to allow for mapping of inheritance hierarchies. Here, nodes are related by a generalisation relationship and the same classes or relations may appear multiple times in different nodes when the data mapping is spread over several objects. Some attributes may also be mapped to constant values, but these values can be overridden in subclass nodes to provide discriminator values between classes.

In the example in Figure 11-2, the person relation appears in three different nodes because it is built up using data from multiple classes. The discriminator column takes a different constant value depending on the most specific class being mapped. Any columns not mapped are given a NULL value by default.

Some mappings may spread a class hierarchy across multiple tables related by a common key value. In these cases, the key value for the subclasses is represented on the arc joining the two nodes. In the example in Figure 11-3, the student.oid (FK) label on the arc connecting the Person and Student nodes denotes that the student relation contains a hidden oid attribute which is a foreign key into the person relation.
11.2 Persistent Store Tests

To evaluate the performance of various persistent stores, I ran a series of tests and recorded the execution time and memory usage of the stores in each case. The results of these tests are summarised in the main report in section 2.1.6 on page 22. This section covers details of how the tests were run, the results recorded and includes source code.

11.2.1 Test Environment

All of the tests were run on the following environment:

<table>
<thead>
<tr>
<th>Model:</th>
<th>IBM Thinkpad</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor:</td>
<td>Intel Pentium M 1500 MHz</td>
</tr>
<tr>
<td>RAM:</td>
<td>512 MB</td>
</tr>
<tr>
<td>Operating System:</td>
<td>Windows XP SP2</td>
</tr>
<tr>
<td>JVM Version:</td>
<td>JDK 1.5.0 Update 6</td>
</tr>
</tbody>
</table>

11.2.2 Results Gathering

For each of the tests, results were gathered in the following way:

- **Execution time** was found by calculating the difference in time in nanoseconds before and after the run method was invoked by using the java `System.nanoTime()` library call. This produces a value based on system time, not CPU time, for the execution of the test code. To reduce the impact made by other processes on the result, these tests were run a number of times and the average time difference was calculated. The actual number of runs is stated for each individual test.

- **Memory usage** was determined by a different method based on the type of store being evaluated. For relational databases, this value was obtained from statistics provided by the DBMS itself. For serialization and object databases, this value was estimated from the size of the database files produced.

11.2.3 Insert Test With Small Objects

This test aimed to measure the execution time and memory overhead of each persistent store when storing small objects (containing less than 1 kB of data each).
Test Code

```java
public void run(int n, IStore store) throws Exception {
    ICarFactory factory = CarFactoryFactory.getCarFactory(store.getDatabase());
    List<ISolenoid> solenoids = new ArrayList<ISolenoid>();
    for (int i = 0; i < 3; i++) {
        ISolenoid s = factory.createSolenoid();
        s.setPartNo(i);
        s.setManufacturer("Some Manufacturer");
        solenoids.add(s);
    }
    List<IStarterMotor> starterMotors = new ArrayList<IStarterMotor>();
    for (int i = 0; i < 5; i++) {
        IStarterMotor s = factory.createStarterMotor();
        s.setPartNo(i);
        s.setManufacturer("Some Manufacturer");
        s.setSolenoid(solenoids.get(i % 3));
        starterMotors.add(s);
    }
    List<IEngine> engines = new ArrayList<IEngine>();
    for (int i = 0; i < 20; i++) {
        IEngine e = factory.createEngine();
        e.setModelNo(i);
        e.setLitres((float)i/5);
        e.setHorsepower(i*20);
        e.setStarterMotor(starterMotors.get(i % 5));
        engines.add(e);
    }
    List<IChassis> chassis = new ArrayList<IChassis>();
    for (int i = 0; i < 10; i++) {
        IChassis c = factory.createChassis();
        c.setModelNo(i);
        c.setManufacturer("Some Manufacturer");
        chassis.add(c);
    }
    for (int i = 0; i < n; i++) {
        ICar c = factory.createCar();
        c.setModelNo(i);
        c.setName("car " + i);
        c.setEngine(engines.get(i % 20));
        c.setChassis(chassis.get(i % 10));
        store.storeCar(c);
    }
}
```

Full code for the tests can be obtained from the project website.
Test Strategy

The test code was executed with values of \( n \) ranging from 100 to 1000 in increments of 100 varying the number of objects stored each time. For execution time, 5 runs of the test were performed for each value of \( n \) and each store.

Test Results

<table>
<thead>
<tr>
<th>Objects</th>
<th>Normal</th>
<th>Serialization</th>
<th>PostgreSQL Basic Mapping</th>
<th>PostgreSQL Single Table Aggregation</th>
<th>PostgreSQL Foreign Key Aggregation</th>
<th>MySQL Basic Mapping</th>
<th>MySQL Single Table Aggregation</th>
<th>MySQL Foreign Key Aggregation</th>
<th>Objectivity</th>
<th>DB4O</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>0.0001</td>
<td>0.0001</td>
<td>0.0004</td>
<td>0.0003</td>
<td>0.0003</td>
<td>5.4973</td>
<td>37.7241</td>
<td>19.5245</td>
<td>11.4577</td>
<td>20.2674</td>
</tr>
<tr>
<td>200</td>
<td>0.0001</td>
<td>0.0001</td>
<td>0.0004</td>
<td>0.0003</td>
<td>0.0003</td>
<td>5.8798</td>
<td>10.3887</td>
<td>19.5245</td>
<td>11.4577</td>
<td>34.9710</td>
</tr>
<tr>
<td>300</td>
<td>0.0001</td>
<td>0.0001</td>
<td>0.0004</td>
<td>0.0003</td>
<td>0.0003</td>
<td>6.3474</td>
<td>10.3887</td>
<td>19.5245</td>
<td>11.4577</td>
<td>52.1590</td>
</tr>
<tr>
<td>400</td>
<td>0.0001</td>
<td>0.0001</td>
<td>0.0004</td>
<td>0.0003</td>
<td>0.0003</td>
<td>6.8366</td>
<td>10.3887</td>
<td>19.5245</td>
<td>11.4577</td>
<td>52.1590</td>
</tr>
<tr>
<td>500</td>
<td>0.0001</td>
<td>0.0001</td>
<td>0.0004</td>
<td>0.0003</td>
<td>0.0003</td>
<td>7.3358</td>
<td>10.3887</td>
<td>19.5245</td>
<td>11.4577</td>
<td>52.1590</td>
</tr>
</tbody>
</table>

Figure 11-4 Execution time in seconds for persisting different numbers of small objects in various persistent stores

![Figure 11-5 Graph of insert performance for small objects](image-url)
11.2.4 Select Test For A Single Small Object

This test aimed to measure the execution time for selecting a single object from the data store. The predicate used to retrieve the object does not require the traversal of any references.

Test Code

This test requests a single object that fulfils a particular predicate from each persistent store. The predicate used tests the value of the modelNo member of the Car class against a particular integer. Since there is no common interface for performing this action across all the persistent stores tested, the majority of this test was custom coded for each different store. Please see the full source code on the project website.

Test Strategy

Before this test was run, each database was pre-filled with 10,000 Car instances using the code from the Insert Test With Small Objects (see section 11.2.3). The test code was then executed with the modelNo of the specific instance being selected ranging from 0 to 10,000 in increments of 1000. For each modelNo value and each persistent store, the test was run 20 times and the average time recorded.

Test Results

<table>
<thead>
<tr>
<th>Object Index</th>
<th>Normal</th>
<th>Serialization</th>
<th>PostgreSQL Basic Mapping</th>
<th>PostgreSQL Single Table Aggregation</th>
<th>PostgreSQL Foreign Key Aggregation</th>
<th>MySQL Basic Mapping</th>
<th>MySQL Single Table Aggregation</th>
<th>MySQL Foreign Key Aggregation</th>
<th>Objectivity</th>
<th>DB4O</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.0250</td>
<td>0.0443</td>
<td>0.0138</td>
<td>0.0139</td>
<td>0.0136</td>
<td>0.0137</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td>235.1866</td>
<td>219.1217</td>
<td>221.1704</td>
<td>218.7616</td>
<td>220.7759</td>
<td>221.0856</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2000</td>
<td>2.2312</td>
<td>2.2300</td>
<td>3.1446</td>
<td>3.7113</td>
<td>4.5004</td>
<td>5.3000</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3000</td>
<td>0.9278</td>
<td>1.4838</td>
<td>2.1940</td>
<td>2.7751</td>
<td>3.4058</td>
<td>4.5204</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4000</td>
<td>1.5641</td>
<td>1.9373</td>
<td>2.5269</td>
<td>3.6641</td>
<td>3.5568</td>
<td>4.0547</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>4.4897</td>
<td>5.1181</td>
<td>8.8916</td>
<td>10.1640</td>
<td>11.2965</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>4.5321</td>
<td>5.0905</td>
<td>6.8967</td>
<td>8.954</td>
<td>7.8211</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>23.4413</td>
<td>26.1272</td>
<td>361.0153</td>
<td>414.8698</td>
<td>449.1618</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>19.2076</td>
<td>19.8296</td>
<td>118.5904</td>
<td>134.3752</td>
<td>148.2392</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 11-6 Execution time in milliseconds for selecting an object with the specified index from various persistent stores.
11.2.5 Select Test For Multiple Small Objects

This test aims to compare the performance of the persistent stores for retrieving multiple objects matching a predicate. The predicate requires the traversal of object references. This tests both the performance of reference traversal and whether each store optimises its search based on the number of objects requested.

Test Code

This test requests a certain number of objects that fulfil a particular predicate from each persistent store. The predicate used tests whether the partNo member variable of the Solenoid instance associated with each car is equal to ‘1’. Since there is no common interface for performing this action across all the persistent stores tested, the majority of this test was custom coded for each different store. Please see the full source code on the project website.

Test Strategy

Before this test was run, each database was pre-filled with 10,000 Car instances using the code from the Insert Test With Small Objects (see section 11.2.3). The test code was then executed with the number of objects requested ranging from 0 to 3000 in increments of 200. For each number of objects requested and each persistent store, the test was run 20 times and the average time recorded.
### Test Results

<table>
<thead>
<tr>
<th>Objects Returned</th>
<th>0</th>
<th>200</th>
<th>400</th>
<th>600</th>
<th>800</th>
<th>1000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Normal</td>
<td>0.0270</td>
<td>0.0152</td>
<td>0.0122</td>
<td>0.0128</td>
<td>0.0125</td>
<td>0.0140</td>
</tr>
<tr>
<td>Serialization</td>
<td>203.6973</td>
<td>221.8457</td>
<td>218.7837</td>
<td>279.2104</td>
<td>277.1328</td>
<td>819.7246</td>
</tr>
<tr>
<td>PostgreSQL Basic Mapping</td>
<td>458.1157</td>
<td>570.8623</td>
<td>637.2089</td>
<td>682.0215</td>
<td>721.6228</td>
<td>819.7246</td>
</tr>
<tr>
<td>PostgreSQL Single Table Aggregation</td>
<td>1.7278</td>
<td>4.5452</td>
<td>12.3136</td>
<td>15.8700</td>
<td>38.4737</td>
<td>32.8378</td>
</tr>
<tr>
<td>PostgreSQL Foreign Key Aggregation</td>
<td>3.0270</td>
<td>183.2004</td>
<td>213.3465</td>
<td>205.0527</td>
<td>221.3850</td>
<td>237.8336</td>
</tr>
<tr>
<td>MySQL Single Table Aggregation</td>
<td>17.1219</td>
<td>47.8310</td>
<td>68.3993</td>
<td>65.5743</td>
<td>94.3502</td>
<td>92.8119</td>
</tr>
<tr>
<td>MySQL Foreign Key Aggregation</td>
<td>12.2694</td>
<td>78.9147</td>
<td>91.6243</td>
<td>99.3207</td>
<td>115.5202</td>
<td>148.0641</td>
</tr>
<tr>
<td>Objectivity</td>
<td>0.7005</td>
<td>50.5014</td>
<td>116.1687</td>
<td>180.0276</td>
<td>185.8705</td>
<td>216.1856</td>
</tr>
<tr>
<td>DB4O</td>
<td>494.8625</td>
<td>506.1391</td>
<td>503.2348</td>
<td>503.3409</td>
<td>522.0919</td>
<td>528.7296</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Objects Returned</th>
<th>1200</th>
<th>1400</th>
<th>1600</th>
<th>1800</th>
<th>2000</th>
<th>2200</th>
</tr>
</thead>
<tbody>
<tr>
<td>Normal</td>
<td>0.0183</td>
<td>0.0132</td>
<td>0.0134</td>
<td>0.0133</td>
<td>0.0135</td>
<td>0.0132</td>
</tr>
<tr>
<td>Serialization</td>
<td>272.1260</td>
<td>277.2273</td>
<td>302.8016</td>
<td>304.0906</td>
<td>328.2897</td>
<td>328.2897</td>
</tr>
<tr>
<td>PostgreSQL Basic Mapping</td>
<td>867.1674</td>
<td>927.5023</td>
<td>1095.1470</td>
<td>1145.7723</td>
<td>1216.6680</td>
<td>1216.6680</td>
</tr>
<tr>
<td>PostgreSQL Single Table Aggregation</td>
<td>28.4221</td>
<td>56.7329</td>
<td>72.0912</td>
<td>101.8621</td>
<td>94.4863</td>
<td>94.4863</td>
</tr>
<tr>
<td>PostgreSQL Foreign Key Aggregation</td>
<td>251.9916</td>
<td>287.2169</td>
<td>300.1518</td>
<td>293.0745</td>
<td>300.4732</td>
<td>300.4732</td>
</tr>
<tr>
<td>MySQL Basic Mapping</td>
<td>842.7547</td>
<td>961.6006</td>
<td>1004.8357</td>
<td>1061.2237</td>
<td>1126.5579</td>
<td>1126.5579</td>
</tr>
<tr>
<td>MySQL Single Table Aggregation</td>
<td>105.6228</td>
<td>139.9709</td>
<td>173.9602</td>
<td>148.6819</td>
<td>186.1446</td>
<td>186.1446</td>
</tr>
<tr>
<td>MySQL Foreign Key Aggregation</td>
<td>202.0210</td>
<td>212.7422</td>
<td>244.1343</td>
<td>241.3064</td>
<td>253.2845</td>
<td>253.2845</td>
</tr>
<tr>
<td>Objectivity</td>
<td>203.3330</td>
<td>315.3529</td>
<td>360.3191</td>
<td>388.0565</td>
<td>427.8030</td>
<td>577.5834</td>
</tr>
<tr>
<td>DB4O</td>
<td>528.7370</td>
<td>561.6662</td>
<td>503.2348</td>
<td>503.3409</td>
<td>522.0919</td>
<td>528.7296</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Objects Returned</th>
<th>2400</th>
<th>2600</th>
<th>2800</th>
<th>3000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Normal</td>
<td>0.0134</td>
<td>0.0136</td>
<td>0.0131</td>
<td>0.0560</td>
</tr>
<tr>
<td>Serialization</td>
<td>323.0636</td>
<td>381.8532</td>
<td>378.2789</td>
<td></td>
</tr>
<tr>
<td>PostgreSQL Basic Mapping</td>
<td>1227.2997</td>
<td>1402.1759</td>
<td>1485.7167</td>
<td></td>
</tr>
<tr>
<td>PostgreSQL Single Table Aggregation</td>
<td>143.4561</td>
<td>145.0681</td>
<td>144.5609</td>
<td></td>
</tr>
<tr>
<td>PostgreSQL Foreign Key Aggregation</td>
<td>298.1156</td>
<td>320.3693</td>
<td>326.1218</td>
<td></td>
</tr>
<tr>
<td>MySQL Basic Mapping</td>
<td>1167.5920</td>
<td>1301.4096</td>
<td>1355.6814</td>
<td></td>
</tr>
<tr>
<td>MySQL Single Table Aggregation</td>
<td>145.2836</td>
<td>180.0650</td>
<td>178.6012</td>
<td></td>
</tr>
<tr>
<td>MySQL Foreign Key Aggregation</td>
<td>279.7644</td>
<td>343.2503</td>
<td>357.0777</td>
<td></td>
</tr>
<tr>
<td>Objectivity</td>
<td>449.7638</td>
<td>547.9666</td>
<td>559.1232</td>
<td></td>
</tr>
<tr>
<td>DB4O</td>
<td>604.5855</td>
<td>657.6140</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Figure 11-8** Execution time in milliseconds for requesting different numbers of objects that fulfil a particular predicate from various persistent stores.

**Figure 11-9** Graph for select performance for multiple objects out of 10,000.
11.2.6 Memory Usage Test With Small Objects

This test aims to assess the memory usage (in secondary storage) of each persistent store for a set of small objects.

Test Strategy

For this test, each persistent store was pre-filled with 10,000 Car instances using the code from the Insert Test With Small Objects (see section 11.2.3). The memory usage for each store was then measured using the method described in section 11.2.2. A second test was then run, this time with half of the generated Car instances having null references for their engine and chassis members.

Test Results

<table>
<thead>
<tr>
<th>Store</th>
<th>Size (kB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Serialization</td>
<td>304</td>
</tr>
<tr>
<td>PostgreSQL Basic Mapping</td>
<td>8000</td>
</tr>
<tr>
<td>PostgreSQL Single Table Aggregation</td>
<td>1544</td>
</tr>
<tr>
<td>PostgreSQL Foreign Key Aggregation</td>
<td>3440</td>
</tr>
<tr>
<td>MySQL Basic Mapping</td>
<td>8500</td>
</tr>
<tr>
<td>MySQL Single Table Aggregation</td>
<td>2000</td>
</tr>
<tr>
<td>MySQL Foreign Key Aggregation</td>
<td>3548</td>
</tr>
<tr>
<td>Objectivity</td>
<td>1832</td>
</tr>
<tr>
<td>DB4O</td>
<td>7861</td>
</tr>
</tbody>
</table>

*Figure 11-10 Results for memory usage test with small objects*

![Memory usage in kilobytes for storing 10,000 small objects](image)

*Figure 11-11 Graph of memory usage in kilobytes for storing 10,000 small objects*
11.2.7 Insert Test With Large Objects

This test aims to measure the execution time of each persistent store when persisting large objects (around 150 kB each).
Test Code

```java
public void run(int n, IStore store) throws Exception {
    IArticleFactory factory = ArticleFactoryFactory.getArticleFactory(store.getDatabase());
    List<IWriter> writers = new ArrayList<IWriter>();
    for (int i = 0; i < 20; i++) {
        IWriter w = factory.createWriter();
        w.setFirstName("Writer");
        w.setLastName(Integer.toString(i));
        w.setPhoto(ImageLoader.loadSmallImage());
        writers.add(w);
    }
    for (int i = 0; i < n; i++) {
        IArticle a = factory.createArticle();
        a.setTitle("Article " + i);
        a.setWriter(writers.get(i % 20));
        a.setId(i);
        a.setImage(ImageLoader.loadLargeImage());
        a.setContent(TextLoader.loadText(i % 10));
        store.storeArticle(a);
    }
}
```

Test Strategy

The test code was executed with values of n ranging from 100 to 1000 in increments of 100 varying the number of objects stored each time. For execution time, 5 runs of the test were performed for each value of n and each store.

Test Results

<table>
<thead>
<tr>
<th>Objects</th>
<th>100</th>
<th>200</th>
<th>300</th>
<th>400</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Normal</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1.4495</td>
<td>3.0162</td>
<td>5.0488</td>
<td>9.0065</td>
</tr>
<tr>
<td></td>
<td>Serialization</td>
<td>30.0868</td>
<td>193.1020</td>
<td>515.3589</td>
</tr>
<tr>
<td></td>
<td>PostgreSQL Basic Mapping</td>
<td>6.0616</td>
<td>18.6983</td>
<td>25.6926</td>
</tr>
<tr>
<td></td>
<td>PostgreSQL Single Table Aggregation</td>
<td>7.7793</td>
<td>19.2399</td>
<td>26.6504</td>
</tr>
<tr>
<td></td>
<td>PostgreSQL Foreign Key Aggregation</td>
<td>9.7405</td>
<td>19.0897</td>
<td>26.6504</td>
</tr>
<tr>
<td></td>
<td>MySQL Basic Mapping</td>
<td>14.0958</td>
<td>34.0115</td>
<td>51.9833</td>
</tr>
<tr>
<td></td>
<td>MySQL Single Table Aggregation</td>
<td>13.8965</td>
<td>28.5808</td>
<td>50.3832</td>
</tr>
<tr>
<td></td>
<td>MySQL Foreign Key Aggregation</td>
<td>22.3955</td>
<td>48.0904</td>
<td>96.1266</td>
</tr>
<tr>
<td></td>
<td>Objectivity</td>
<td>28.4342</td>
<td>67.2991</td>
<td>80.1331</td>
</tr>
<tr>
<td></td>
<td>DB4O</td>
<td>36.3158</td>
<td>68.6203</td>
<td>106.6548</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Objects</th>
<th>500</th>
<th>600</th>
<th>700</th>
<th>800</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Normal</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>10.5967</td>
<td>12.7229</td>
<td>16.0399</td>
<td>16.9288</td>
</tr>
<tr>
<td></td>
<td>Serialization</td>
<td>43.0012</td>
<td>48.0094</td>
<td>63.5117</td>
</tr>
<tr>
<td></td>
<td>PostgreSQL Basic Mapping</td>
<td>40.6122</td>
<td>53.8658</td>
<td>71.6215</td>
</tr>
<tr>
<td></td>
<td>PostgreSQL Single Table Aggregation</td>
<td>42.0263</td>
<td>45.8444</td>
<td>57.1591</td>
</tr>
<tr>
<td></td>
<td>PostgreSQL Foreign Key Aggregation</td>
<td>115.6983</td>
<td>126.4216</td>
<td>160.4569</td>
</tr>
<tr>
<td></td>
<td>MySQL Basic Mapping</td>
<td>88.7840</td>
<td>98.8366</td>
<td>115.2793</td>
</tr>
<tr>
<td></td>
<td>MySQL Single Table Aggregation</td>
<td>132.2993</td>
<td>176.7689</td>
<td>196.4938</td>
</tr>
<tr>
<td></td>
<td>MySQL Foreign Key Aggregation</td>
<td>236.5583</td>
<td>289.6507</td>
<td>262.3270</td>
</tr>
</tbody>
</table>

**Figure 11-14** Execution time in seconds for persisting different numbers of large objects in various persistent stores
Only limited results were gathered for serialization and DB4O due to heap space errors that could not be avoided. Serialization generated errors at 400 objects, while DB4O generated errors at 900.

11.2.8 Select Test For A Single Large Object (1)

This test aims to assess the performance of selecting a single object from a store of large objects. In this test, only small fields of the large object are actually accessed.

**Test Code**

This test requests only the title field of a single, specific object that fulfils a particular predicate from each persistent store. The predicate used tests whether the id member variable of an Article instance is equal to a specified value. Since there is no common interface for performing this action across all the persistent stores tested, the majority of this test was custom coded for each different store. Please see the full source code on the project website.

**Test Strategy**

Before this test was run, each database was pre-filled with 1000 Article instances using the code from the Insert Test With Large Objects (see section 11.2.7). The test code was then executed with the id of the specific instance being selected ranging from 0 to 1000 in increments of 100. For each id value and each persistent store, the test was run 20 times and the average time recorded. Serialization and DB4O were omitted from this test due to memory errors when pre-filling them with data.
Appendix

Test Results

<table>
<thead>
<tr>
<th>Object</th>
<th>0</th>
<th>100</th>
<th>200</th>
<th>300</th>
<th>400</th>
<th>500</th>
</tr>
</thead>
<tbody>
<tr>
<td>Normal</td>
<td>0.0507</td>
<td>0.0579</td>
<td>0.0504</td>
<td>0.0122</td>
<td>0.0120</td>
<td>0.0114</td>
</tr>
<tr>
<td>PostgreSQL</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Basic Mapping</td>
<td>2.3123</td>
<td>1.4554</td>
<td>1.5526</td>
<td>3.4155</td>
<td>2.7328</td>
<td>2.7291</td>
</tr>
<tr>
<td>PostgreSQL</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Single Table</td>
<td>0.9491</td>
<td>0.9034</td>
<td>1.0901</td>
<td>4.5085</td>
<td>2.4717</td>
<td>4.6364</td>
</tr>
<tr>
<td>Aggregation</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PostgreSQL</td>
<td>1.4690</td>
<td>1.3691</td>
<td>1.5534</td>
<td>2.5175</td>
<td>3.7754</td>
<td>3.6218</td>
</tr>
<tr>
<td>MySQL</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Basic Mapping</td>
<td>34.8597</td>
<td>90.7520</td>
<td>93.4288</td>
<td>78.6222</td>
<td>80.1568</td>
<td>2339.6813</td>
</tr>
<tr>
<td>PostgreSQL</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Single Table</td>
<td>11.8857</td>
<td>58.0823</td>
<td>79.6214</td>
<td>66.0794</td>
<td>74.9750</td>
<td>2665.6148</td>
</tr>
<tr>
<td>Aggregation</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PostgreSQL</td>
<td>11.0427</td>
<td>57.5298</td>
<td>74.0003</td>
<td>73.8901</td>
<td>37.1111</td>
<td>2464.4135</td>
</tr>
<tr>
<td>MySQL</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Basic Mapping</td>
<td>2968.1848</td>
<td>3416.4392</td>
<td>3972.7289</td>
<td>4437.5952</td>
<td>4698.3819</td>
<td>4698.5212</td>
</tr>
<tr>
<td>PostgreSQL</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Single Table</td>
<td>3280.1235</td>
<td>3723.7481</td>
<td>4199.8422</td>
<td>4772.3442</td>
<td>5315.4947</td>
<td>5315.4947</td>
</tr>
<tr>
<td>Aggregation</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MySQL</td>
<td>3006.1582</td>
<td>3579.7808</td>
<td>3998.7097</td>
<td>4547.6530</td>
<td>5039.2282</td>
<td>5039.2282</td>
</tr>
<tr>
<td>Objectivity</td>
<td>296.3569</td>
<td>296.4834</td>
<td>291.4839</td>
<td>303.4320</td>
<td>295.3322</td>
<td>295.3322</td>
</tr>
</tbody>
</table>

Figure 11-16 Execution time in milliseconds for selecting the smallest field from a single large object from various persistent stores

Figure 11-17 Graph of select performance for a single large object from 1000
11.2.9 Select Test For A Single Large Object (2)

This test aims to assess the performance of selecting a single object from a store of large objects. In this test, the largest field (the `image` field) of the object is requested.

Test Code

This test requests the `image` field of a single, specific object that fulfils a particular predicate from each persistent store. The code is unmodified from the previous test, except for the field that is selected.

Test Strategy

The test strategy was identical to that of the previous test.

Test Results

<table>
<thead>
<tr>
<th>Object Index</th>
<th>0</th>
<th>100</th>
<th>200</th>
<th>300</th>
<th>400</th>
<th>500</th>
</tr>
</thead>
<tbody>
<tr>
<td>Normal</td>
<td>0.4240</td>
<td>0.0230</td>
<td>0.0221</td>
<td>0.0215</td>
<td>0.0216</td>
<td>0.0216</td>
</tr>
<tr>
<td>PostgreSQL Basic Mapping</td>
<td>56.6239</td>
<td>55.4967</td>
<td>54.0599</td>
<td>54.5337</td>
<td>51.5779</td>
<td>51.3422</td>
</tr>
<tr>
<td>PostgreSQL Single Table Aggregation</td>
<td>56.2972</td>
<td>93.2156</td>
<td>94.3822</td>
<td>100.2752</td>
<td>94.7901</td>
<td>90.9018</td>
</tr>
<tr>
<td>PostgreSQL Foreign Key Aggregation</td>
<td>60.1108</td>
<td>93.5532</td>
<td>95.3844</td>
<td>91.0718</td>
<td>89.9626</td>
<td>98.6450</td>
</tr>
<tr>
<td>MySQL Basic Mapping</td>
<td>17.4009</td>
<td>1044.4770</td>
<td>2158.1714</td>
<td>3103.4278</td>
<td>4228.7372</td>
<td>5274.9876</td>
</tr>
<tr>
<td>MySQL Single Table Aggregation</td>
<td>17.1746</td>
<td>1119.6562</td>
<td>2377.8890</td>
<td>3420.1571</td>
<td>4582.1647</td>
<td>5775.1517</td>
</tr>
<tr>
<td>MySQL Foreign Key Aggregation</td>
<td>18.5038</td>
<td>902.3477</td>
<td>1737.6785</td>
<td>2668.3334</td>
<td>3764.0485</td>
<td>4741.4766</td>
</tr>
<tr>
<td>Objectivity</td>
<td>188.0689</td>
<td>293.0015</td>
<td>293.7462</td>
<td>295.1670</td>
<td>293.4037</td>
<td>289.5370</td>
</tr>
</tbody>
</table>

Figure 11-18 Execution time in milliseconds for selecting the largest field of a single large object from various persistent stores
Select Test For Multiple Large Objects

This test measures the execution time of fetching multiple large objects from each persistent store.

Test Code

This test requests a certain number of objects that fulfil a particular predicate from each persistent store. The predicate used tests whether the lastName member variable of the Writer instance associated with each article is equal to ‘1’. Since there is no common interface for performing this action across all the persistent stores tested, the majority of this test was custom coded for each different store. Please see the full source code on the project website.

Test Strategy

Before this test was run, each database was pre-filled with 1000 Article instances using the code from the Insert Test With Large Objects (see section 11.2.7). The test code was then executed with the number of objects requested ranging from 0 to 300 in increments of 20. For each number of objects requested and each persistent store, the test was run 20 times and the average time recorded.
Test Results

<table>
<thead>
<tr>
<th>Objects Returned</th>
<th>Normal</th>
<th>PostgreSQL Basic Mapping</th>
<th>PostgreSQL Single Table Aggregation</th>
<th>PostgreSQL Foreign Key Aggregation</th>
<th>MySQL Basic Mapping</th>
<th>MySQL Single Table Aggregation</th>
<th>MySQL Foreign Key Aggregation</th>
<th>Objectivity</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>0.0278 0.0241 2.8785 0.0321 0.0650 0.0117 0.0605 0.0126</td>
<td>1413.5344 1455.392 1457.9405 1494.3120 1528.7845 1415.4381 1425.9817 1474.5622</td>
<td>4.7603 10.2742 7.7959 18.7369 15.7964 18.5165 18.4237 19.8387</td>
<td>10022.9391 10239.8124 10261.0351 10276.8424 10292.5754 10320.2503 10306.6584 10307.8192</td>
<td>18.6660 253.9897 499.9675 841.7756 1218.7241 1627.0416 1877.0707 2179.9797</td>
<td>24.1510 451.7210 846.4211 1221.8943 1612.6402 2068.5425 2489.1055 2923.9974</td>
<td>1.3305 1582.1176 3178.3710 4900.6301 6400.6632 8264.6851 9820.5554 11783.5742</td>
</tr>
</tbody>
</table>

Figure 11-20 Execution time in milliseconds for selecting multiple large objects from various persistent stores

<table>
<thead>
<tr>
<th>Objects Returned</th>
<th>Normal</th>
<th>PostgreSQL Basic Mapping</th>
<th>PostgreSQL Single Table Aggregation</th>
<th>PostgreSQL Foreign Key Aggregation</th>
<th>MySQL Basic Mapping</th>
<th>MySQL Single Table Aggregation</th>
<th>MySQL Foreign Key Aggregation</th>
<th>Objectivity</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>160</td>
<td>0.0771 0.0129 0.0590 0.0127 0.0592 0.0124 0.0622 0.0129</td>
<td>1467.64212 1571.6759 1546.2772 1613.4626 1639.7292 1622.5220 1635.7362 1688.4224</td>
<td>11.474028 7.4952 8.2243 8.8692 13.4676 13.9602 11.3284 50.2369</td>
<td>22.4648104 19.836 17.8572 18.9922 19.0925 19.6812 20.7186 20.7166</td>
<td>10314.4485 10343.8147 10343.7449 10529.7780 10387.5731 10472.1873 10514.5572 10398.4584</td>
<td>3339.82699 3727.6301 4288.7581 4578.7243 5019.1650 5454.1522 5837.4487 6210.2780</td>
<td>13217.4419 1485.1276 15522.9892 1680.4250 18240.2549 19524.9212 20595.5727 21856.8469</td>
</tr>
</tbody>
</table>

Figure 11-21 Graph of select performance for multiple large objects out of 1000
11.2.11  Insert Test For Class Hierarchies

This test evaluates the performance of each store at persisting objects that are part of a class hierarchy consisting of five classes.

Test Code

```java
public void run(int n, IStore store) throws Exception {
    IPersonFactory factory =
    PersonFactoryFactory.getPersonFactory(store.getDatabase());
    for (int i = 0; i < n; i++) {
        IPerson p = null;
        if (i % 4 == 0) {
            p = factory.createPerson();
            p.setFirstName("Person");
        } else if (i % 4 == 1) {
            IStaff s = factory.createStaff();
            s.setFirstName("Staff");
            s.setPay(i*1000);
            p = s;
        } else if (i % 4 == 2) {
            IOverseasStudent s = factory.createOverseasStudent();
            s.setFirstName("OverseasStudent");
            s.setYear(i % 4 % 4);
            s.setCountry("Somewhere");
            p = s;
        } else if (i % 4 == 3) {
            IHomeStudent s = factory.createHomeStudent();
            s.setFirstName("HomeStudent");
            s.setYear(i % 4 % 4);
            if (i % 4 % 3 == 0) {
                s.setLea("Bristol");
            } else {
                s.setLea("Some LEA");
            }
            p = s;
        }
        p.setLastName(Integer.toString(i));
        p.setAge(20 + (i % 3));
        store.storePerson(p);
    }
}
```
Balancing simplicity and efficiency in Web applications

Full code for the tests can be obtained from the project website.

**Test Strategy**

The test code was executed with values of n ranging from 100 to 1000 in increments of 100 varying the number of objects stored each time. For execution time, 5 runs of the test were performed for each value of n and each store.

**Test Results**

<table>
<thead>
<tr>
<th>Objects</th>
<th>100</th>
<th>200</th>
<th>300</th>
<th>400</th>
<th>500</th>
</tr>
</thead>
<tbody>
<tr>
<td>Normal</td>
<td>0.0002</td>
<td>0.0002</td>
<td>0.0003</td>
<td>0.0003</td>
<td>0.0005</td>
</tr>
<tr>
<td>Serialization</td>
<td>0.5228</td>
<td>1.1288</td>
<td>2.0794</td>
<td>3.4627</td>
<td>5.9203</td>
</tr>
<tr>
<td>PostgreSQL Table Per Hierarchy</td>
<td>0.9070</td>
<td>0.3630</td>
<td>0.3992</td>
<td>0.5007</td>
<td>0.6133</td>
</tr>
<tr>
<td>PostgreSQL Table Per Path</td>
<td>0.7533</td>
<td>0.9889</td>
<td>0.2785</td>
<td>0.3556</td>
<td>0.4325</td>
</tr>
<tr>
<td>PostgreSQL Table Per Class</td>
<td>1.3343</td>
<td>2.2926</td>
<td>0.8834</td>
<td>1.0332</td>
<td>1.2209</td>
</tr>
<tr>
<td>MySQL Table Per Hierarchy</td>
<td>9.9226</td>
<td>13.3953</td>
<td>19.6023</td>
<td>26.1553</td>
<td>32.3744</td>
</tr>
<tr>
<td>MySQL Table Per Path</td>
<td>10.9237</td>
<td>13.4281</td>
<td>20.1287</td>
<td>25.7602</td>
<td>32.3945</td>
</tr>
<tr>
<td>MySQL Table Per Class</td>
<td>21.4854</td>
<td>26.6808</td>
<td>40.7907</td>
<td>56.1865</td>
<td>79.0731</td>
</tr>
<tr>
<td>Objectivity</td>
<td>29.2216</td>
<td>28.0415</td>
<td>55.3871</td>
<td>69.7090</td>
<td>88.5485</td>
</tr>
<tr>
<td>DB4O</td>
<td>48.2385</td>
<td>49.6339</td>
<td>55.3871</td>
<td>69.7090</td>
<td>88.5485</td>
</tr>
</tbody>
</table>

**Figure 11-22** Execution time in seconds for persisting different numbers of objects from a class hierarchy in various persistent stores

**Figure 11-23** Graph of insert performance for class hierarchies

180
11.2.12 Select Test For Multiple Objects From A Class Hierarchy (1)

The aim of this test is to determine the performance of each store when retrieving multiple objects that fulfil some condition over the superclass fields.

Test Code

This test requests a certain number of objects that fulfil a particular predicate from each persistent store. The predicate used tests whether the age member variable of each Person instance is equal to ‘20’. Since there is no common interface for performing this action across all the persistent stores tested, the majority of this test was custom coded for each different store. Please see the full source code on the project website.

Test Strategy

Before this test was run, each database was pre-filled with 10,000 Person instances using the code from the Insert Test For Class Hierarchies (see section 11.2.11). The test code was then executed with the number of objects requested ranging from 0 to 3000 in increments of 200. For each number of objects requested and each persistent store, the test was run 20 times and the average time recorded.

Test Results

<table>
<thead>
<tr>
<th>Objects Returned</th>
<th>0</th>
<th>200</th>
<th>400</th>
<th>600</th>
<th>800</th>
<th>1000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Normal</td>
<td>0.0438</td>
<td>0.0251</td>
<td>0.0206</td>
<td>0.0193</td>
<td>0.0215</td>
<td>0.0209</td>
</tr>
<tr>
<td>Serialization</td>
<td>347.7352</td>
<td>330.6072</td>
<td>366.7192</td>
<td>368.6391</td>
<td>362.5827</td>
<td>390.1247</td>
</tr>
<tr>
<td>PostgreSQL Table Per Hierarchy</td>
<td>3.1316</td>
<td>10.7666</td>
<td>17.0493</td>
<td>20.2779</td>
<td>29.0888</td>
<td>37.8791</td>
</tr>
<tr>
<td>PostgreSQL Table Per Path</td>
<td>2.3671</td>
<td>112.6499</td>
<td>121.2044</td>
<td>128.4767</td>
<td>147.2657</td>
<td>153.8649</td>
</tr>
<tr>
<td>PostgreSQL Table Per Class</td>
<td>2.2901</td>
<td>10.0809</td>
<td>17.0493</td>
<td>20.2779</td>
<td>29.0888</td>
<td>37.8791</td>
</tr>
<tr>
<td>MySQL Table Per Hierarchy</td>
<td>16.9052</td>
<td>27.1878</td>
<td>40.2792</td>
<td>55.9152</td>
<td>71.4141</td>
<td>98.6416</td>
</tr>
<tr>
<td>MySQL Table Per Path</td>
<td>231.4567</td>
<td>237.4664</td>
<td>279.6951</td>
<td>295.0417</td>
<td>322.3522</td>
<td>343.4920</td>
</tr>
<tr>
<td>MySQL Table Per Class</td>
<td>15.4416</td>
<td>37.6898</td>
<td>35.9831</td>
<td>55.3260</td>
<td>83.3203</td>
<td>95.5657</td>
</tr>
<tr>
<td>Objectivity</td>
<td>1.2790</td>
<td>32.7018</td>
<td>60.3841</td>
<td>99.1600</td>
<td>118.4224</td>
<td>168.5342</td>
</tr>
<tr>
<td>DB4O</td>
<td>1893.3540</td>
<td>1917.1195</td>
<td>1916.3868</td>
<td>1939.2787</td>
<td>1945.0112</td>
<td>1852.6764</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Objects Returned</th>
<th>1200</th>
<th>1400</th>
<th>1600</th>
<th>1800</th>
<th>2000</th>
<th>2200</th>
</tr>
</thead>
<tbody>
<tr>
<td>Normal</td>
<td>0.0215</td>
<td>0.0207</td>
<td>0.0213</td>
<td>0.0206</td>
<td>0.0204</td>
<td>0.0204</td>
</tr>
<tr>
<td>Serialization</td>
<td>382.1122</td>
<td>395.4582</td>
<td>396.0185</td>
<td>392.8746</td>
<td>425.9990</td>
<td>485.1548</td>
</tr>
<tr>
<td>PostgreSQL Table Per Hierarchy</td>
<td>43.9741</td>
<td>48.3998</td>
<td>59.4259</td>
<td>60.3914</td>
<td>169.1931</td>
<td></td>
</tr>
<tr>
<td>PostgreSQL Table Per Path</td>
<td>185.2114</td>
<td>198.1103</td>
<td>201.7434</td>
<td>231.4283</td>
<td>203.9011</td>
<td>260.2836</td>
</tr>
<tr>
<td>PostgreSQL Table Per Class</td>
<td>52.7554</td>
<td>106.8988</td>
<td>124.3551</td>
<td>115.4218</td>
<td>159.5128</td>
<td>170.2214</td>
</tr>
<tr>
<td>MySQL Table Per Hierarchy</td>
<td>124.6452</td>
<td>122.2790</td>
<td>127.2176</td>
<td>165.4508</td>
<td>177.0160</td>
<td>162.3058</td>
</tr>
<tr>
<td>MySQL Table Per Path</td>
<td>367.4443</td>
<td>363.0952</td>
<td>403.9817</td>
<td>431.7824</td>
<td>448.4174</td>
<td>445.0474</td>
</tr>
<tr>
<td>MySQL Table Per Class</td>
<td>113.4702</td>
<td>126.2713</td>
<td>124.8413</td>
<td>130.2547</td>
<td>165.9118</td>
<td>194.3649</td>
</tr>
<tr>
<td>Objectivity</td>
<td>193.8283</td>
<td>229.8232</td>
<td>240.0248</td>
<td>287.3851</td>
<td>290.2874</td>
<td>477.8146</td>
</tr>
<tr>
<td>DB4O</td>
<td>1893.3540</td>
<td>1917.1195</td>
<td>1916.3868</td>
<td>1939.2787</td>
<td>1945.0112</td>
<td>1852.6764</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Objects Returned</th>
<th>2400</th>
<th>2600</th>
<th>2800</th>
<th>3000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Normal</td>
<td>0.0209</td>
<td>0.0213</td>
<td>0.0214</td>
<td>0.0210</td>
</tr>
<tr>
<td>Serialization</td>
<td>468.9733</td>
<td>482.2957</td>
<td>476.4226</td>
<td>482.9753</td>
</tr>
<tr>
<td>PostgreSQL Table Per Hierarchy</td>
<td>79.3256</td>
<td>121.8230</td>
<td>164.3982</td>
<td>198.7189</td>
</tr>
<tr>
<td>PostgreSQL Table Per Path</td>
<td>237.5776</td>
<td>297.3863</td>
<td>279.1483</td>
<td>333.3028</td>
</tr>
<tr>
<td>PostgreSQL Table Per Class</td>
<td>223.3105</td>
<td>253.2362</td>
<td>213.1132</td>
<td>224.6017</td>
</tr>
<tr>
<td>MySQL Table Per Hierarchy</td>
<td>223.0736</td>
<td>206.6329</td>
<td>253.6141</td>
<td>272.9693</td>
</tr>
<tr>
<td>MySQL Table Per Path</td>
<td>533.8273</td>
<td>526.4384</td>
<td>533.3008</td>
<td>570.2295</td>
</tr>
<tr>
<td>MySQL Table Per Class</td>
<td>232.8359</td>
<td>213.6165</td>
<td>214.5481</td>
<td>228.2989</td>
</tr>
<tr>
<td>Objectivity</td>
<td>363.2249</td>
<td>505.5681</td>
<td>477.4496</td>
<td>597.4132</td>
</tr>
<tr>
<td>DB4O</td>
<td>1748.0018</td>
<td>1858.4795</td>
<td>1884.2861</td>
<td>1812.4865</td>
</tr>
</tbody>
</table>

Figure 11-24 Execution time in milliseconds for requesting different numbers of objects that fulfil a particular superclass predicate from various persistent stores
The aim of this test is to determine the performance of each store when retrieving multiple objects that fulfil some condition over the fields of a specific subclass.

**Test Code**

This test requests a certain number of objects that fulfil a particular predicate from each persistent store. The predicate used tests whether the `lea` member variable of each `HomeStudent` instance is equal to “Bristol”. Since there is no common interface for performing this action across all the persistent stores tested, the majority of this test was custom coded for each different store. Please see the full source code on the project website.

**Test Strategy**

Before this test was run, each database was pre-filled with 10,000 `Person` instances using the code from the Insert Test For Class Hierarchies (see section 11.2.11). The test code was then executed with the number of objects requested ranging from 0 to 3000 in increments of 200. For each number of objects requested and each persistent store, the test was run 20 times and the average time recorded.
## Test Results

<table>
<thead>
<tr>
<th>Objects</th>
<th>0</th>
<th>200</th>
<th>400</th>
<th>600</th>
<th>800</th>
<th>1000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Normal</td>
<td>0.0478</td>
<td>0.0238</td>
<td>0.0214</td>
<td>0.0199</td>
<td>0.0205</td>
<td>0.0217</td>
</tr>
<tr>
<td>Serialization</td>
<td>386.5793</td>
<td>397.4904</td>
<td>465.7046</td>
<td>487.1235</td>
<td>424.5952</td>
<td>489.5394</td>
</tr>
<tr>
<td>PostgreSQL Table Per Hierarchy</td>
<td>2.3900</td>
<td>13.6530</td>
<td>15.5832</td>
<td>23.8587</td>
<td>34.0297</td>
<td>35.7524</td>
</tr>
<tr>
<td>PostgreSQL Table Per Path</td>
<td>2.7298</td>
<td>20.2752</td>
<td>24.8035</td>
<td>31.0411</td>
<td>84.2959</td>
<td>60.1552</td>
</tr>
<tr>
<td>MySQL Table Per Class</td>
<td>2.5742</td>
<td>22.1977</td>
<td>42.2396</td>
<td>70.4539</td>
<td>60.7499</td>
<td>86.4177</td>
</tr>
<tr>
<td>Normal</td>
<td>0.0203</td>
<td>0.0308</td>
<td>0.0220</td>
<td>0.0224</td>
<td>0.0214</td>
<td>0.0222</td>
</tr>
<tr>
<td>Serialization</td>
<td>479.1304</td>
<td>422.3519</td>
<td>432.6207</td>
<td>423.3483</td>
<td>424.4107</td>
<td>484.0268</td>
</tr>
<tr>
<td>PostgreSQL Table Per Hierarchy</td>
<td>129.9279</td>
<td>92.7839</td>
<td>135.2458</td>
<td>118.4583</td>
<td>119.9320</td>
<td>142.8238</td>
</tr>
<tr>
<td>PostgreSQL Table Per Path</td>
<td>107.4415</td>
<td>152.6239</td>
<td>149.8334</td>
<td>198.9434</td>
<td>193.1771</td>
<td>199.7075</td>
</tr>
<tr>
<td>MySQL Table Per Class</td>
<td>113.4416</td>
<td>130.2252</td>
<td>148.9338</td>
<td>166.1189</td>
<td>192.2793</td>
<td>198.3647</td>
</tr>
<tr>
<td>Normal</td>
<td>0.0220</td>
<td>0.0308</td>
<td>0.0220</td>
<td>0.0224</td>
<td>0.0214</td>
<td>0.0222</td>
</tr>
<tr>
<td>Serialization</td>
<td>479.1304</td>
<td>422.3519</td>
<td>432.6207</td>
<td>423.3483</td>
<td>424.4107</td>
<td>484.0268</td>
</tr>
<tr>
<td>PostgreSQL Table Per Hierarchy</td>
<td>129.9279</td>
<td>92.7839</td>
<td>135.2458</td>
<td>118.4583</td>
<td>119.9320</td>
<td>142.8238</td>
</tr>
<tr>
<td>PostgreSQL Table Per Path</td>
<td>107.4415</td>
<td>152.6239</td>
<td>149.8334</td>
<td>198.9434</td>
<td>193.1771</td>
<td>199.7075</td>
</tr>
<tr>
<td>MySQL Table Per Class</td>
<td>113.4416</td>
<td>130.2252</td>
<td>148.9338</td>
<td>166.1189</td>
<td>192.2793</td>
<td>198.3647</td>
</tr>
<tr>
<td>Normal</td>
<td>0.0220</td>
<td>0.0308</td>
<td>0.0220</td>
<td>0.0224</td>
<td>0.0214</td>
<td>0.0222</td>
</tr>
<tr>
<td>Serialization</td>
<td>479.1304</td>
<td>422.3519</td>
<td>432.6207</td>
<td>423.3483</td>
<td>424.4107</td>
<td>484.0268</td>
</tr>
<tr>
<td>PostgreSQL Table Per Hierarchy</td>
<td>129.9279</td>
<td>92.7839</td>
<td>135.2458</td>
<td>118.4583</td>
<td>119.9320</td>
<td>142.8238</td>
</tr>
<tr>
<td>PostgreSQL Table Per Path</td>
<td>107.4415</td>
<td>152.6239</td>
<td>149.8334</td>
<td>198.9434</td>
<td>193.1771</td>
<td>199.7075</td>
</tr>
<tr>
<td>MySQL Table Per Class</td>
<td>113.4416</td>
<td>130.2252</td>
<td>148.9338</td>
<td>166.1189</td>
<td>192.2793</td>
<td>198.3647</td>
</tr>
<tr>
<td>Objectivity</td>
<td>1.2392</td>
<td>30.1696</td>
<td>31.7759</td>
<td>49.6986</td>
<td>108.5073</td>
<td>98.2025</td>
</tr>
<tr>
<td>DB4O</td>
<td>421.7607</td>
<td>407.9260</td>
<td>455.5966</td>
<td>462.7355</td>
<td>534.3113</td>
<td>573.2337</td>
</tr>
</tbody>
</table>

**Figure 11-26** Execution time in milliseconds for requesting different numbers of objects that fulfil a particular subclass predicate from various persistent stores

![Select performance for multiple class hierarchy objects from 10,000](image-url)

**Figure 11-27** Graph of select performance for multiple class hierarchy objects from 10,000
11.2.14 Memory Usage Test With Class Hierarchies

The aim of this test is to assess the memory usage (in secondary storage) of each persistent store for a set of objects that belong to a class hierarchy.

Test Strategy

Before this test was run, each database was pre-filled with 10,000 Person instances using the code from the Insert Test For Class Hierarchies (see section 11.2.11). The test code was then executed with the number of objects requested ranging from 0 to 3000 in increments of 200. For each number of objects requested and each persistent store, the test was run 20 times and the average time recorded.

Test Results

![Figure 11-28 Graph of memory usage in kilobytes for 10,000 class hierarchy objects](image-url)
11.3 Case Study Tests

To analyse the web application used in the case study, the P6Spy\(^{29}\) query logger was installed which would log all database queries executed by the application. I then wrote an additional utility that split the log file up into separate per-request files and would record the total duration of each request. This utility was run on several pages for each module being tested and then analysed to detect fine and coarse query granularity problems (see section 3.1.2). The following is detailed analysis of each module tested.

11.3.1 Announcements Module

The announcements module is one of the simplest modules in the web application and consists of a message board on which users can post announcements and view announcements posted within the same community (or parent community). Announcements are stored within a single entity bean class (AnnouncementBean) which defines the following fields:

```java
public abstract Long getId();
public abstract String getAuthor();
public abstract String getTitle();
public abstract byte[] getContent();
public abstract String getCommunity();
public abstract Date getDate();
public abstract Date getExpire();
public abstract int getPriority();
public abstract long getLastModified();
public abstract boolean getPublished();
public abstract long getWorkflowId();
```

They use container managed persistence, which provides access to the following finders via calls to the AnnouncementModule session bean:

```java
Collection findByCommunityAfter(java.lang.String community, java.util.Date expire, boolean published):
SELECT OBJECT(o) FROM Announcement o WHERE o.community=?1 AND o.expire > ?2 AND o.published = ?3

Collection findNonPublishedWorkflow(java.lang.String community, boolean published):
SELECT OBJECT(o) FROM Announcement o WHERE o.community=?1 AND o.workflowId>0 AND o.published = ?2

Collection findPublished(boolean published):
SELECT OBJECT(o) FROM Announcement o WHERE o.published = ?1

Collection findByCommunityBefore(java.lang.String community, java.util.Date expire, boolean published):
SELECT OBJECT(o) FROM Announcement o WHERE o.community=?1 AND o.expire < ?2 AND o.published = ?3

Collection findAll():
SELECT OBJECT(o) from Announcement o
com.formicary.epix.module.announce.Announcement findByWorkflowId(java.lang.Long id):
SELECT OBJECT(o) from Announcement o WHERE o.workflowId = ?1
```

\(^{29}\) P6Spy is an open-source query logger that works with a variety of J2EE-based application servers. It can be downloaded from http://www.p6spy.com/.
Balancing simplicity and efficiency in Web applications

Added 100 announcements to module, the index page showed all 100 entries, producing the following table:

<table>
<thead>
<tr>
<th>Title</th>
<th>Expiry Date</th>
<th>Priority</th>
<th>Written By</th>
<th>Posted On</th>
<th>Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>This is another test announcement</td>
<td>28 Nov 05</td>
<td>Low</td>
<td>Admin User</td>
<td>21 Nov 05 18:04</td>
<td>▶️ ◄ ✗</td>
</tr>
<tr>
<td>This is another test announcement</td>
<td>28 Nov 05</td>
<td>Low</td>
<td>Admin User</td>
<td>21 Nov 05 18:04</td>
<td>▶️ ◄ ✗</td>
</tr>
<tr>
<td>This is another test announcement</td>
<td>28 Nov 05</td>
<td>Low</td>
<td>Admin User</td>
<td>21 Nov 05 18:04</td>
<td>▶️ ◄ ✗</td>
</tr>
<tr>
<td>This is another test announcement</td>
<td>28 Nov 05</td>
<td>Low</td>
<td>Admin User</td>
<td>21 Nov 05 18:04</td>
<td>▶️ ◄ ✗</td>
</tr>
<tr>
<td>This is another test announcement</td>
<td>28 Nov 05</td>
<td>Low</td>
<td>Admin User</td>
<td>21 Nov 05 18:03</td>
<td>▶️ ◄ ✗</td>
</tr>
<tr>
<td>This is another test announcement</td>
<td>28 Nov 05</td>
<td>Low</td>
<td>Admin User</td>
<td>21 Nov 05 18:03</td>
<td>▶️ ◄ ✗</td>
</tr>
</tbody>
</table>

*Figure 11-29 Announcements index table*

This page also generated the following query in the logs:

```
SELECT o1.id, o1.author, o1.title, o1.content, o1.community, o1.date_, o1.expire, o1.priority, o1.lastModified, o1.published, o1.workflowId FROM module_announce_item o1 WHERE o1.community = 'home' AND (o1.expire > '2005-11-20 13:32:41.791' AND o1.published = 'true')
```

This seems a likely candidate for the coarse query problem, since it requests all columns of the announcements table despite the fact that only five seem to be displayed on the page. Looking at the code, the announcements are fetched from a J2EE container (AnnouncementModuleBean) which contains a method for calling the `findByCommunityAfter` finder:

```java
public Collection getCurrentAnnouncements(String community) {
    Collection announcements = new HashSet();
    ...
    announcements.addAll(getAnnouncementHome().findByCommunityAfter(c, now, true));
    ...
    return getView(community, announcements, false);
}
```

This finder is declared as follows:

```
Collection findByCommunityAfter(java.lang.String community, java.util.Date expire, boolean published):
SELECT OBJECT(o) FROM Announcement o WHERE o.community=?1 AND o.expire > ?2 AND o.published = ?3
```

This generates a query that selects all columns from the announcements table because the developer is expecting entire Announcement instances to be returned. The querying system has no knowledge of how the instances are to be used, so it must fetch all the (non-relation) fields so that the data is available in case it is needed.
Alternatively, the developer could specify in the finder exactly which fields are to be used, but this feature is not supported in EJB2. Even if it were supported, it would require the developer to balance two opposing requirements:

- The fields selected should be as close to the usage as possible in order to improve efficiency.
- The query should be as general as possible to allow for re-use.

There is also another problem found when tracing through the code. At the end of the container method, a `getView(..)` method is called:

```java
private Collection getView(String communityName, Collection announcements, boolean checkLimit)
...
List data = new ArrayList(announcements.size());
Iterator announcementsIterator = announcements.iterator();
while(announcementsIterator.hasNext())
{
    Announcement announcement = (Announcement)announcementsIterator.next();
    data.add(announcement.getData());
}
Collections.sort(data, new Comparator()
{
    public int compare(Object o1, Object o2)
    {
        ... date comparison ...
    }
});
return new ArrayList(data.subList(0, limit));
```

In J2EE, often information is sent over RMI calls so that the beans for providing content and the actions for displaying it can be on separate machines. The J2EE guidelines recommend converting all objects retrieved from the database into Data objects before returning them from the container in order to reduce the number of RMI calls made. This means that all objects in the returned collection are almost guaranteed to be accessed, if only to be converted to data objects, restricting the ability to optimise fetching of objects from the database by delaying queries.

Also worthy of note is the sorting of the collection of announcements after they have been converted into data objects as well as the application of a limit (the `data.subList(..)` call at the end). Since there is no processing on the data that needs to be performed in the Java domain before these occur (with the exception of the conversion to data objects), we could migrate all of this to the SQL query so that it could be performed on the database.

Continuing the code trace, the above container method is called from an action class (`ViewAllAction`), which obtains the list of announcements and makes it available to the JSP:

```java
announcements = ((AnnounceModule)getModule()).getCurrentAnnouncements(getCommunity().getName());
```

The JSP then reads the announcements and generates the table in the page:

```html
<ww:bean name="'webwork.util.Sorter'" id="sorter"/>
<ww:sort source="announcements"
comparator="@sorter/descending('priority')">
    <ww:iterator status="'status'" id="idAnnounce">
        ... table-building code ...
    </ww:iterator>
</ww:sort>
```
This highlights another problem; the container already sorted the list chronologically before it was returned and it is now being sorted by priority. Again, this could be migrated to the query and possibly combined into a single sort to avoid traversing the entire collection twice.

All of these optimisations were manually applied to the code and the service times for both the optimised and unoptimised versions were profiled for varying numbers of announcements. The following results were obtained:

<table>
<thead>
<tr>
<th>Announcements</th>
<th>Unoptimised</th>
<th>Optimised</th>
<th>Announcements</th>
<th>Unoptimised</th>
<th>Optimised</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.5877</td>
<td>0.3024</td>
<td>2100</td>
<td>9.0369</td>
<td>3.2529</td>
</tr>
<tr>
<td>100</td>
<td>0.6966</td>
<td>0.6427</td>
<td>2200</td>
<td>9.1040</td>
<td>3.4094</td>
</tr>
<tr>
<td>200</td>
<td>1.2170</td>
<td>0.4980</td>
<td>2300</td>
<td>9.2839</td>
<td>3.4587</td>
</tr>
<tr>
<td>300</td>
<td>1.7238</td>
<td>0.6390</td>
<td>2400</td>
<td>10.5210</td>
<td>3.5364</td>
</tr>
<tr>
<td>400</td>
<td>2.0760</td>
<td>0.7936</td>
<td>2500</td>
<td>10.7135</td>
<td>3.6769</td>
</tr>
<tr>
<td>500</td>
<td>2.3840</td>
<td>0.8540</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>600</td>
<td>2.9441</td>
<td>1.1195</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>700</td>
<td>3.0760</td>
<td>1.2099</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>800</td>
<td>3.5008</td>
<td>1.2744</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>900</td>
<td>4.0616</td>
<td>1.3022</td>
<td>2700</td>
<td>12.8758</td>
<td>4.7480</td>
</tr>
<tr>
<td>1000</td>
<td>4.2068</td>
<td>1.5624</td>
<td>2800</td>
<td>12.5461</td>
<td>4.4992</td>
</tr>
<tr>
<td>1100</td>
<td>4.6264</td>
<td>1.6483</td>
<td>2900</td>
<td>12.2252</td>
<td>4.7035</td>
</tr>
<tr>
<td>1200</td>
<td>5.2927</td>
<td>1.7395</td>
<td>3000</td>
<td>12.6732</td>
<td>4.7743</td>
</tr>
<tr>
<td>1300</td>
<td>5.5775</td>
<td>1.9250</td>
<td>3100</td>
<td>13.7640</td>
<td>4.8726</td>
</tr>
<tr>
<td>1400</td>
<td>6.0919</td>
<td>2.0279</td>
<td>3200</td>
<td>13.3366</td>
<td>4.9416</td>
</tr>
<tr>
<td>1500</td>
<td>6.3102</td>
<td>2.4688</td>
<td>3300</td>
<td>13.7002</td>
<td>4.5024</td>
</tr>
<tr>
<td>1600</td>
<td>6.7387</td>
<td>2.3299</td>
<td>3400</td>
<td>13.9025</td>
<td>4.5384</td>
</tr>
<tr>
<td>1700</td>
<td>7.1525</td>
<td>2.5112</td>
<td>3500</td>
<td>13.7201</td>
<td>4.5872</td>
</tr>
<tr>
<td>1800</td>
<td>7.5372</td>
<td>2.5468</td>
<td>3600</td>
<td>13.8579</td>
<td>4.8704</td>
</tr>
<tr>
<td>1900</td>
<td>8.3539</td>
<td>2.6607</td>
<td>3700</td>
<td>14.9324</td>
<td>4.8421</td>
</tr>
</tbody>
</table>

Figure 11-30 Request service time in seconds for the unoptimised and optimised announcements index page

Figure 11-31 Graph of announcements index page service times
The calendar module allows users to post calendar events which last a certain duration and view events from the same community (or parent community). Events can be viewed on a per-day, per-week, per-month and per-year basis. This analysis is mainly concerned with the month view; the other views all exhibit very similar behaviour. Calendar events are stored across two associated entity beans. The first (EventBean) stores information about each event:

```java
public abstract Long getId();
public abstract String getOwner();
public abstract String getCommunityName();
public abstract boolean getIsPublic();
public abstract String getOccurrence();
public abstract int getNumOccurrences();
public abstract Date getStartDate();
public abstract long getDuration();
public abstract long getSequenceID();
public abstract String getEventType();
public abstract String getTitle();
public abstract String getDescription();
public abstract long getAssociatedEventID();
public abstract long getLastModified();
public abstract boolean getIsAllDayEvent();
```
It also specifies the following finders:

```
Collection findAll():
  SELECT DISTINCT OBJECT(o) FROM Event o

Collection findBetween(java.lang.String community, java.util.Date start, java.util.Date end):
  SELECT DISTINCT OBJECT(o) FROM Event o WHERE o.communityName=?1 AND o.startDate > ?2
     AND o.startDate < ?3

Collection findBetween(java.util.Date start, java.util.Date end):
  SELECT DISTINCT OBJECT(o) FROM Event o WHERE o.startDate > ?1 AND o.startDate < ?2

Collection findByCommunity(java.lang.String community):
  SELECT DISTINCT OBJECT(o) FROM Event o WHERE o.communityName=?1

Collection findByOwner(java.lang.String owner):
  SELECT DISTINCT OBJECT(o) FROM Event o WHERE o.owner=?1
```

The second (CalendarDayBean) stores information about each day in the calendar:

```
public abstract int getDay();
public abstract int getMonth();
public abstract int getYear();
public abstract Set getEvents();
```

It also specifies these finders:

```
Collection findAll():
  SELECT DISTINCT OBJECT(o) FROM CalendarDay o

Collection findDaysInMonth(int day1, int day2, int month, int year):
  SELECT DISTINCT OBJECT(o) FROM CalendarDay o WHERE o.day >= ?1 AND o.day <= ?2
     AND o.month = ?3 AND o.year= ?4

Collection findByYear(int year):
  SELECT DISTINCT OBJECT(o) FROM CalendarDay o WHERE o.year= ?1

Collection findBetween(int day1, int month1, int year1, int day2, int month2, int year2):
  SELECT DISTINCT OBJECT(o) FROM CalendarDay o WHERE (o.year > ?3 OR (o.year = ?3 AND
     (o.month > ?2 OR (o.day >= ?1 AND o.month = ?2)))) AND (o.year < ?6 OR (o.year = ?6
     AND (o.month < ?5 OR (o.day <= ?4 AND o.month = ?5))))
```

The two are associated by the `getEvents()` method in CalendarDayBean, which stores a Set of IDs of events that start on that day. Each of the bean classes has their own table (module_calendar_event and module_calendar_day respectively) and the association is stored in a separate lookup table (module_calendar_day_events).

Added 10 calendar entries over varying dates, the month view showed all 10 entries producing the following page:
Several queries appear in the logs for this page. First, all the CalendarDay entities for the month are obtained:

```sql
SELECT o1.day, o1.month, o1.year
FROM module_calendar_day o1
WHERE o1.day >= 1
AND (o1.day <= 31
AND (o1.month = 1
AND o1.year = 2006))
```

Next, for each day, the association table is queried to find the IDs of all events that start on that day:

```sql
SELECT value
FROM calendar_day_events
WHERE (day = 1
AND month = 1
AND year = 2006)
```

Finally, each event that starts on that day is requested separately:

```sql
SELECT module_calendar_event.owner, module_calendar_event.communityName,
module_calendar_event.isPublic, module_calendar_event.occurrence,
module_calendar_event.numOccurrences, module_calendar_event.startDate,
module_calendar_event.duration, module_calendar_event.sequenceID,
module_calendar_event.eventType, module_calendar_event.title,
module_calendar_event.description, module_calendar_event.associatedEventID,
module_calendar_event.lastModified, module_calendar_event.isAllDayEvent
FROM module_calendar_event
WHERE (module_calendar_event.id = 20)
```

![Figure 11-33 Calendar month view](image)
Balancing simplicity and efficiency in Web applications

This looks likely to exhibit fine query granularity, since many queries are executed each retrieving only a single object. Looking at the code, the announcements are fetched from a J2EE container (CalendarModuleBean) which contains a method for calling the findDaysInMonth finder from CalendarDayBean:

```java
public Map getEventsByMonth(String communityName, boolean useInheritance, int day1, int day2, int month, int year) throws FinderException
{
    Map results = new HashMap();
    Collection communities = getCommunities(communityName, useInheritance);
    Collection monthEvents = getCalendarDayHome().findDaysInMonth(day1, day2, month, year);
    ...
}
```

The code then iterates through the returned CalendarDay entities, calling the getEvents() method on each:

```java
for (Iterator i = monthEvents.iterator(); i.hasNext();)
{
    CalendarDay day = (CalendarDay)i.next();
    List events = (List)results.get(new Integer(day.getDay()));
    ...
    Collection dayEvents = day.getEvents();
}
```

Then, the event IDs returned are iterated through, each event is obtained by calling the findByPrimaryKey finder from EventBean, checked to see if it belongs to a visible community and, if so, its data object is added to a list:

```java
Iterator j = dayEvents.iterator();
while (j.hasNext())
{
    Long eventId = (Long)j.next();
    Event event = getEventHome().findByPrimaryKey(eventId);
    if (communities.contains(event.getCommunityName()))
    {
        events.add(event.getData());
    }
}
```

Finally, the events are sorted by start date:

```java
Collections.sort(events, new EventComparator());
```

This exhibits many of the optimisation opportunities of the announcements module, such as field selection (all fields of each event are obtained, but only community, startDate, duration, eventType and title are needed), conversion to data objects and sorting in the Java domain. It also highlights one new one; eager loading. For reasons explained in section 2.2.5, eager loading is often a very bad strategy, but ideally the references that are going to be traversed should be loaded eagerly and the others not. In this case, lazy loading is used for the events associated with a particular day, which causes several separate queries to be executed which could be replaced by a single, coarse grained query that performs a JOIN across the three tables (in fact, only two tables are required since the module_calendar_day data is actually duplicated in the module_calendar_day_events table):

```sql
SELECT module_calendar_events.day, module_calendar_event.communityname, module_calendar_event.startdate, module_calendar_event.duration, module_calendar_event.eventtype, module_calendar_event.title
FROM module_calendar_event, module_calendar_day_events
WHERE module_calendar_day_events.day <= 31
AND module_calendar_day_events.month = 1
AND module_calendar_day_events.year = 2006
AND module_calendar_event.id = module_calendar_day_events.value
```
AND module_calendar_event.communityname IN ( ... results of getCommunities(..) ... )
ORDER BY module_calendar_event.startdate ASC

This module was manually optimised to use the above query and page service times for the month view were profiled across a range of 0-1000 calendar entries in intervals of 100. For each number of entries, a separate process mimicking a web browser made ten requests for both the optimised and unoptimised pages, timed each using Java’s nanoTime() standard library call, then averaged the times to produce the final results.

<table>
<thead>
<tr>
<th>Events</th>
<th>Unoptimised</th>
<th>Optimised</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1.1829</td>
<td>0.3577</td>
</tr>
<tr>
<td>100</td>
<td>8.7407</td>
<td>0.8103</td>
</tr>
<tr>
<td>200</td>
<td>16.7376</td>
<td>1.1715</td>
</tr>
<tr>
<td>300</td>
<td>24.8946</td>
<td>1.6726</td>
</tr>
<tr>
<td>400</td>
<td>33.0326</td>
<td>2.5938</td>
</tr>
<tr>
<td>500</td>
<td>43.1178</td>
<td>2.2665</td>
</tr>
<tr>
<td>600</td>
<td>48.6179</td>
<td>2.7451</td>
</tr>
<tr>
<td>700</td>
<td>55.7016</td>
<td>3.2415</td>
</tr>
<tr>
<td>800</td>
<td>64.2694</td>
<td>3.6414</td>
</tr>
<tr>
<td>900</td>
<td>72.2487</td>
<td>4.0546</td>
</tr>
<tr>
<td>1000</td>
<td>82.3356</td>
<td>4.5494</td>
</tr>
</tbody>
</table>

**Figure 11-34** Request service time in seconds for unoptimised and optimised versions of the calendar module

**Figure 11-35** Graph of calendar month view service times
11.4 Requirements Questionnaire

To gather the opinions of web developers on subjects relating to this project, I constructed an online questionnaire which ran over a four week period and gathered responses from 21 developers. This section includes results and any comments left for each of the questions; for analysis of the results, please see section 3.2 of the main report. The questionnaire itself is still online, although no longer gathering results, and is available from the project website.

When constructing the questionnaire, the following areas were considered:\(^{30}\):

- Each question should be multiple-choice since it ensures that results are easily categorised.
- A comments box should exist for each question, so that additional responses and comments can be left.
- The questions should be worded to encourage a wide range of results and not biased to any particular answer.
- All “pick one” questions should have options that cover the full range of possible responses so that all users should be able to answer it.

\(^{30}\) Many of these were taken from an online tutorial on professional questionnaire writing, which can be found at http://www.statpac.com/surveys/.
Which of the following technologies have you used to write websites?

![Bar chart showing technology usage]

**Figure 11-37** Results for “Which of the following technologies have you used to write websites?”

**Additional Comments:**
- Asp, coldfusion also.
- Tcl
- JavaScript!
- Javascript?
- I don’t want to give the idea that I used JSP in the website because I didn’t - servlets all the way. I’ve realised that developing for servlets and JSPs are two different games.

This question was designed to determine the range of abilities of those answering the questionnaire. The large number of hits for the languages used to create dynamic websites is encouraging, since the questionnaire is mainly concerned with dynamic sites that need to store data.
How large a team do you typically work in when developing websites?

![Pie chart showing the distribution of team sizes. I work alone, 7, 35%; 2-3 people, 7, 35%; 4-9 people, 5, 25%; 10-20 people, 1, 5%; 20+ people, 0, 0%]

**Figure 11-38** Results for “How large a team do you typically work in when developing websites?”

**Additional Comments:**
- I have also worked for 6 months in a professional website design company which employs around 20 developers.
- That is not by choice; I prefer less (3)
- JavaScript!

This question was included because some of the answers to later questions are likely to be biased depending on whether the user works in a team or not. If hardly any of those who answered the questionnaire had worked in a team, the results of these later questions would likely be worthless, but the good spread of results obtained indicates that this is not the case.
Do you prefer to use an object-oriented approach when writing websites?

![Pie chart showing the results of the survey question: Do you prefer to use an object-oriented approach when writing websites?]

*Figure 11-39* Results for “Do you prefer to use an object-oriented approach when writing websites?”

**Additional Comments:**
- Yes since I’m quite the wiz.
- JavaScript!
- This is one of the worst questions ever!
- Well, it depends on what “object oriented” means in this context - I definitely organize a model and use it, but I typically don’t use component-oriented technologies like Tapestry.

This question aimed to help determine the target language of this project. The large response in favour of object-oriented languages is probably coupled with the large number of respondents who work in a team, and therefore probably develop enterprise applications for which the common procedural web languages (PHP, Perl) are not suitable.
Do you prefer to use a statically-typed language when writing websites?

![Pie chart showing preferences:
- Yes: 10, 50%
- No: 4, 20%
- Don't mind: 4, 20%
- Don't know: 2, 10%]

**Figure 11-40** Results for “Do you prefer to use a statically-typed language when writing websites?”

**Additional Comments:**
- JavaScript!
- Equally bad. On the grounds that you need HTML and HTML isn’t statically typed, everyone uses non-statically typed languages.

Like the question before it, this one was also intended to help narrow down the choice of target language for the project. The strong result in favour of statically-typed languages is probably linked with good IDE support that helps make the code base of large, team-based applications more manageable. In response to the comment on using HTML, although it is true that HTML isn’t statically typed, people do not necessarily prefer to use it over other languages when writing dynamic websites.
Please put the following in order of importance to you when writing websites

This question aimed to help prioritise the requirements for this project. Consistent site design was rated the highest of the options, and so will be an important feature of the presentation layer. Short page load times was also rated highly, and will form the focus of the presentation layer (since this is where the largest performance hit is expected; see section 3.1). Extensibility and maintainability will crosscut all parts of this project, but will probably be focussed in the web server, since this will allow the dynamic loading and updating of classes that will allow for independently designed modules to be deployed and tested separately from one another.

Short development times and thorough testing were seen as reasonably important, so hope to be addressed by clear interfaces, the lack of scaffolding code needed for persistence and a presentation layer that can easily support end-to-end testing. Finally, graphical impact and providing a large number of features were least important so will only be addressed by this project if time allows.
If your websites need to store data, which of the following systems have you used?

![Bar chart showing data storage system usage](image)

**Figure 11-42** Results for "If your websites need to store data, which of the following systems have you used?"

**Additional Comments:**
- Where’s EJB? JDO? etc.
- sybase, oracle
- sybase, oracle
- JavaScript!
- sybase
- oracle

This question aimed to determine which types of data store are most popular, firstly to support the fact that object-oriented databases aren’t widely used (as mentioned in section 2.1.4) and also to see what types of systems people are used to using. Ideally, users should be able to easily switch from the most popular systems to the product produced by this project. Since the results of this question strongly support the use of relational databases, this should be an achievable goal. To address the comment on the absence of Enterprise Java Beans and Java Data Objects from the list of possible answers, although they are valid methods for storing data, they are simply layers on top of some other secondary storage system. Oracle and Sybase were omitted purely due to space reasons.
How would you prefer to store data for your site, assuming efficiency is not an issue?

![Diagram](image)

**Figure 11-43** Results of "How would you prefer to store data for your site, assuming efficiency is not an issue?"

**Additional Comments:**
- Are we talking data as in textural or binary? Either way a relational database.
- JavaScript!
- I don’t care how it’s stored. I only care about the ways in which I can access it.
- I’d prefer to use a standard repository, for example, JCR.
- Which ever way is fastest to code and takes the least investment in understanding is preferable for me. (Since if it is complex to understand, it is probably going to be hard to make correct)

This question was intended to determine what kind of interface users would prefer to use when storing and retrieving data, however the comments indicate that it should have been made less ambiguous. It was expected that most users who prefer to use an object-oriented language to write web applications would also prefer to use an object database to store their data, but in this case such users may have chosen a relational database because of the more standard interfaces and querying mechanisms it offers. More work is needed on this before any concrete conclusions can be drawn.
If you have used a relational database in the past, do you ensure that your table schema are optimised for efficient data access (by defining indices...etc.)?

**Figure 11-44** Results for “If you have used a relational database in the past, do you ensure that your table schema are optimised for efficient data access (by defining indices...etc.)?”

**Additional Comments:**
- No since I’m a ejb whore.
- JavaScript!
- People who use normalised RDBMs and don’t define foreign key constraints should be shot.

This question intended to ascertain how important table optimisation is in web development. Since a large proportion (70%) of those who answered this questionnaire spend time optimising table schema, it should be addressed by my project by automating the process where possible.
If you have used a relational database in the past, do you ensure that your queries are optimised for efficient data access (by using limits...etc.)?

![Pie chart showing the results of the question](chart.png)

**Figure 11-45** Results for “If you have used a relational database in the past, do you ensure that your queries are optimised for efficient data access (by using limits...etc.)?”

**Additional Comments:**
- Again I let the ejb container take care of it.
- JavaScript!
- Um, normally it’s the best thing to get your SQL right such that you need do as little application level processing of the data as possible.
- I only use limits for reasons of coding simplicity - it is easier than writing a function to select just the range of results that you want. (In general, it is always easier to write using the declarative SQL syntax than writing custom code to select what you want)

In a similar vein to the previous question, this attempted to find out how important query optimisation is in web development. Since this question gathered an even stronger response, this project should also address the problem of automated query optimisation.
If automatic storage of data was efficient, would you still want to manually influence how the data is stored?

**Figure 11-46** Results for "If automatic storage of data was efficient, would you still want to manually influence how the data is stored?"

Additional Comments:
- Only anal fools would want this.
- But it better be damn efficient! (I doubt this will happen any time soon)
- JavaScript!
- No system which is “automatic” will ever get it right all of the time.
- For me, simplicity is important. If automatic storage means lots of complexity and thought involved to make it all work right, it would be a turnoff.

This question was intended to determine whether, if an automated persistence mechanism was implemented, only the automated mapping would need to be implemented or whether an additional API for allowing users to customise the mapping would also be needed. Since there are still a number of people who would want manual control over the mapping, this should be considered if time allows.
Do your websites need to offer their content in multiple different formats (i.e. HTML, RSS, WAP...etc.)?

![Pie chart showing results]

Figure 11-47 Results for "Do your websites need to offer their content in multiple different formats (i.e. HTML, RSS, WAP...etc.)?"

**Additional Comments:**

- JavaScript!

Often, presentation systems seem to be tied to the format which they output to (typically HTML). This question was designed to see whether a more general system that separated out content from format would be useful. With the majority of respondents (65%) saying that their sites do need to output content in different formats, the ability to automatically convert a page into a number of different formats might reduce the development time spent producing separate pages for each different format or writing transformations to convert one format into another. However, more work needs to be done to determine which formats are most widely required in web applications.
Is it important that your websites are compliant with HTML or XHTML specifications?

![Pie chart showing the results of the question: Yes, 17, 85%; No, 3, 15%; Don't know, 0, 0%]

Figure 11-48 Results for "Is it important that your websites are compliant with HTML or XHTML specifications?"

Additional Comments:

- JavaScript!
- Following the specifications as far as possible is sensible. Though sometimes one has to add invalidate code to work around various browser quirks.

One possible feature that could be implemented in the presentation layer is to enforce the use of a particular specification for each supported format. Once the content for each page has been finalised, a formatter could traverse the content and output it in a format that fulfils the necessary specification. The response to this question would indicate that such a feature would be helpful, although the additional comment about browser incompatibility raises an important issue. The developer should be able to override any restriction placed on the outputted content in case a particular browser does not itself support the specification properly.
When adding functionality to your sites, do you prefer to mix writing code to generate content with writing code for creating the graphical interface?

This question intended to see whether developers prefer to use systems that encourage a mix of presentation and content (PHP, JSP) or prefer to generate content in a separate stage and then decorate it with the necessary graphics. The result against mixing the two styles suggests that this issue should be a major consideration when designing the presentation layer.
If you work in a team, are the tasks of working on functionality (PHP, Java...etc.) and presentation (HTML, CSS...etc.) given to different people?

![Pie chart showing the results of the survey question](image)

**Figure 11-50** Results for “If you work in a team, are the tasks of working on functionality (PHP, Java...etc.) and presentation (HTML, CSS...etc.) given to different people?”

Additional Comments:
- And rightfully so.
- JavaScript!
- Unfortunately I have not worked in a large team developing a site. Though when I have worked with other designers and coders: code/design is one of the splits. Though there is usually another split within presentation, between the designers themselves and the HTML/CSS experts.

This question is similar to the last, but aimed to determine the degree of separation between the content and graphical components of each page. If presentation is worked on completely separately to content, then some plugin system whereby presentation classes may be developed independently of the rest of the system and deployed dynamically into running web applications may be appropriate. On the other hand, for those who do not require this much separation, such a system could prove too cumbersome. Since there is no definitive answer to this question, the presentation layer should seek to either support both systems or find a suitable compromise.
Additional Comments

As well as allowing users to add comments to each question individually, the final question asked for additional comments about the questionnaire as a whole. This elicited the following responses:

- I am hungry, can somebody cook lunch for me?
- Use J2EE.