An Evaluation of the Performance and Database Access Strategies of Java Object-Relational Mapping Frameworks

BY

Kevin R. Higgins

Submitted to the Department of Electrical Engineering and Computer Science and the Faculty of the Graduate School of the University of Kansas in partial fulfillment of the requirements for the degree of Master's of Science

Committee Members:

Hossein Saiedian, Ph.D.
Professor
Thesis Chairperson

Arvin Agah, Ph.D.
Associate Professor

Xue-wen Chen, Ph.D.
Assistant Professor

Date Defended: October 5, 2007

The Thesis Committee for Kevin R Higgins
certifies that this is the approved version of the following thesis:

An Evaluation of the Performance and
Database Access Strategies of Java
Object-Relational Mapping Frameworks

Committee Members: _____
Thesis Chairperson

_____

_____

Date Approved: _____

Abstract

With an object-relational mapping (ORM) tool, the software developer does not have direct control over the structured query language (SQL) calls to the database and thus relies on ORM decisions regarding how the database model is accessed. These database access strategies are critical to an application's performance since databases are typically remote from the application's runtime. In essence, an ORM can introduce performance issues if the ORM is not able to generate efficient SQL, manage transactions, and provide an adequate caching mechanism.

This research formulated an ORM performance evaluation framework by defining a set of performance test cases that are based upon common database access scenarios. The research then developed an implementation of the framework using a selected number of Java ORMs in order to evaluate the ORM processing time, database access calls, and the use of object caching within the ORM. The selected ORMs included two Java Persistence API (JPA) ORMs, Hibernate and OpenJPA, and two Java Data Objects (JDO) ORMs, JPOX and Speedo.

The performance test cases revealed significant performance differences between the selected ORMs. The JPA implementations were better performers when compared to the JDO implementations, with Hibernate being the overall best performer of all ORMs. The ability of an ORM to create more complex SQL provided significant performance gains when eager loading objects, bulk loading related objects, and performing batch inserts. The test framework also indicated that caching had a dramatic impact on reducing processing time when multiple calls are made to retrieve objects. In fact, caching was as important as efficient SQL generation when evaluating the processing times.

The results indicate that ORMs exhibit considerable differences in terms of performance and database access, while all executing the same set of test cases. The application of a common set of benchmark performance test cases, as defined and implemented in this research, can be undertaken by the ORM community in order to provide an objective process for evaluating ORM performance. Such a benchmark would provide ORM users with insight into how the ORMs implemented common data access scenarios. This information would then allow developers to better select ORMs for their respective application context.

Acknowledgements

I would like to extend my sincere gratitude to Professor Hossein Saiedian for his invaluable guidance during the course of my graduate studies and in the development of my thesis. I would like to give special thanks to my committee members, Professor Arvin Agah and Professor Xue-wen Chen, for their time in serving on my committee and their useful feedback. Finally, I would like to thank my friends and family for their support during this undertaking.

Table of Contents

List of Tables

List of Figures

1 Introduction

1.1 Background Information

The relational database management system (RDMS) is at the heart of the modern enterprise systems that run businesses today. RDMSs offer flexible and robust ways to manage data, share data between different applications, and secure the information that runs nearly all companies. The technology strategies of companies recognize this fact, as most businesses have large-scale investments in relatively-expensive RDMSs that house the data that runs the business. As such, most information technology (IT) software developers work with some form of RDMS, and it would be difficult to find an IT job that did not involve accessing RDMSs in some fashion.

While software developers find themselves working directly with RDMS on a daily basis, the logical structure of the data in an RDMS is quite different when compared to the object-oriented (OO) languages that are the implementation choice for most IT systems today. An OO language allows the developer to build an object-view of a business since it provides a way to group data and behavior together in a single, logical unit. Complex business logic needs such an OO domain model that offers inheritance and polymorphism, in addition to the design pattern opportunities that OO languages provide. However, while OO languages are

easy to work with regards to business modeling, the OO paradigm is in contrast to the RDMS's representation of data in tabular form where data is logically portrayed in tables and rows.

This difference between the OO paradigm that is used to develop business logic and the RDMS paradigm that is used to store and manage the business data has been called the object-relational paradigm mismatch, also known as *impedance mismatch* (Cattell, 1991). The object-relational mismatch is based on a number of reasons, including the fact that objects come in a range of granularity from coarse-grained to fine-grained, while databases offer just two levels of granularity: table and column. In addition, an RDMS does not provide any mechanism for inheritance or dealing with subtypes. Finally, RDMS associations are represented as foreign keys which are not necessarily directional, while object associations are represented as references that are directional.

Overall, the object-relational mismatch must be solved by the developer if the system is to be maintainable and responsive to change. Solving the problem can be undertaken by interacting directly with the low-level driver interface provided by the RDMS vendor. In this way, a developer manually maps objects to tables by hand-coding structured query language (SQL) calls to the RDMS in order to access and manage the stored data. In addition, the developer manages the communication between the

application and the RDMS, including establishing connections, scrolling through result sets, and handling exceptions. In short, the developer manages all interaction with the database.

The problem with the low-level, call-interface interaction with the database is that it takes time away from developing code that actually solves the business problem at hand. Thus, the developer must code/test/debug code just to store and retrieve data in addition to writing the business logic of the application. As a result, object-to-database applications can require a great deal of coding just to overcome the object-relational impedance mismatch.

An alternative to this low-level access to a RDMS is the use of an object-relational mapping (ORM) tool that hides the RDMS access management and the tabular view of the data. ORMs reduce development cost since the ORM product implements the object-table mapping instead of the developer. In general, the ORM tools provide a code-generated bridge between the object paradigm and the relational paradigm. The ORM manages the database access, scrolls through the result sets, and sets/gets data into the objects that are used in the business logic of the application. While the software developer must still know how to configure the communication between the relational database and the application's

object model, the amount of code to be developed, tested, and debugged is significantly reduced.

1.2 Research Objectives and Methodology

While an ORM offers a large amount in the way of developing manageable code, the removal of the developer from the database access does have its disadvantages. Most notably, the developer does not have direct control over the SQL calls to the database and thus must rely on ORM decisions regarding how the data model is accessed. This database access, consisting of disk read/writes and (typically) remote access, is a key performance consideration in most applications using an RDMS. Due to the performance considerations of RDMS access and the loss of control over that access through the use of an ORM, the choice of an ORM should consider how the ORM has been designed to manage performance. Thus, a comparison of ORMs utilizing a benchmark test suite is required in order to provide an objective evaluation and comparison of ORM performance.

This research developed an ORM performance evaluation framework by defining set of performance test cases that are based upon common database access scenarios. The research then formulated an application of the framework on a selected number of Java ORMs in order to evaluate and compare ORM processing time, database access calls, and the use of object caching within the ORM. The selected test ORMs included the Java

Persistence API (JPA) ORMs, Hibernate and OpenJPA, and the Java Data Objects (JDO) ORMs, JPOX and Speedo.

The performance test cases are built around common database access scenarios that would frequently occur in an information-driven application. These test cases included the following:

- Loading an object twice within the same transaction;
- Loading an object twice between transactions;
- Loading only some of an object's data;
- Lazy-loading an object graph;
- Eager loading an object graph;
- Bulk loading similar objects;
- Bulk saving objects; and,
- Loading and saving a complex object.

The performance test case evaluation was undertaken by developing a benchmark application data model and creating a database schema to represent the storage of that model. The data model contained common entity-relationship multiplicities found in all databases, including one-to-one, one-to-many, and the use of a join table to break a many-to-many relationship into two, one-to-many relationships. The database entities, along with their interrelationships, were mapped to Java objects using each selected ORM. Database access code for the various performance test cases was then developed using each ORM, and the test cases were executed utilizing a testing framework.

1.3 Evaluation Criteria

The evaluation of ORM performance was undertaken by executing a select set of performance test cases. While these test cases do not define *all* possible database access scenarios, the use of a simple set of test cases provides a practical approach that can be used to objectively evaluate performance and database access strategies. Thus, the test cases provide the necessary information required to ascertain whether there are drastic variations between ORMs with regard to processing time and remote database calls.

The evaluation of each ORM was undertaken by measuring the processing time and number of database access calls required to execute each performance test case. These evaluation criteria were selected as these measures provide the two key aspects of database access performance. Processing time reflects the speed of the ORM in processing the task (performance test case), and thus provides a common metric for comparing the ORMs. The number of database access calls, while not directly measuring performance, provides insight into the relative ability of each ORM to perform well in a distributed environment. As the number of calls increase, the ORM performance can be negatively impacted in a distributed architecture simply due to the network overhead with remote database access.

The use of memory by the ORM was not included in the evaluation criteria. Although memory metrics can provide a possible measure of why an ORM performs well or performs poorly, the measure of memory by itself is not an actual measure of performance. The count of the database access calls, in addition to the test case processing time, was considered more valuable in the evaluation as enterprise applications are typically distributed in multi-tiered applications.

1.4 Thesis Organization

The thesis is organized into seven chapters as follows:

1 - Introduction: This chapter presents background information on the importance of ORM frameworks and the objectives of this research.

2 - Java ORM Implementations: This chapter provides a review of Java ORM implementations currently utilized in the industry. Included is a review of standard-based ORMs as well as non-standard-based ORMs.

3 - Assessing ORM Performance: This chapter provides a review of related research into the performance testing of ORM frameworks, including relational databases and object databases.

4 - A Performance Evaluation Framework: This chapter describes the performance testing approach utilized in this research, including the performance test cases, application data model, as well as the software and hardware utilized in the tests.

5 - ORM Performance Results: This chapter presents the results of the performance test cases, including tabular data of all tests and figures of selected tests.

6 - Analysis and Discussion of Results: This chapter provides a full discussion of the performance test case results, investigates the

variances in performance, and discusses the impact of the performance differences on application design decisions.

7 - Conclusions and Future Work: This chapter presents a summary of the thesis, overall conclusions from the performance tests, and outlines future work that could be undertaken in the area of Java ORM performance evaluations.

2 Java ORM Frameworks

2.1 Overview

There are many Java ORM frameworks that are available today, including standard-based ORMs and non-standard-based ORMs. The intent of all of these frameworks is to provide an object-relational solution that frees the developer from writing database access code in addition to eliminating the need to develop an in-house object-relational solution. While proprietary in-house ORM solutions can be developed, such frameworks usually come at a high cost compared to commercial solutions and open-source frameworks.

There are currently three Java standard-based ORM frameworks: Enterprise Java Bean's Container Managed Persistence (EJB-CMP), Java Data Objects (JDO), and the Java Persistence API (JPA). These ORMs are considered "standard-based" since their respective interfaces are based on Java Specification Request (JSR) specifications developed under the Java Community Process (JCP). JSR is a formal process administered by Sun Microsystems, with input from industry experts and vendors of Java products. Thus, these ORMs are integrated into an actual Java release with implementations being supported by both open-source and commercial vendors.

Non-standard-based ORMs are not based on any JSR under the JCP. Instead, these ORMs are based upon a framework and API that has been developed by either an open-source project or a commercial vendor. Thus, implementations of these non-standard ORMs cannot be swapped out with other implementations as their interface, and associated API, is proprietary in nature. Nonetheless, many of these solutions provide feature-rich APIs and stable releases that are backed by a large development community.

2.2 Enterprise Java Beans

Enterprise Java Beans (EJB) is the Sun Microsystems specification for building and managing server-side components for enterprise-wide applications (Sun Microsystems, 2002, 2003, 2006). The specification, which is part of the Java Enterprise Edition (EE), has gone through several releases since it was first created in 1997, with the latest release being EJB 3.0. The overall intent of the standard is to provide a common approach to implementing business logic across an enterprise. This includes APIs for business components, persistence, transaction processing, messaging, naming and directory services, remote procedure calls, and web services.

The EJB design principles are centered on the idea that applications should be loosely coupled and that all EJB behavior is specified by interfaces. The calling applications do not manage resources; instead, the container provides support to the developer. EJB applications are also

tiered, with the session tier representing the API to the application and the entity tier representing the API to the datastore.

The tiered notion of EJBs means they are developed as one of three types: Session Beans, Message Driven Beans, and Entity Beans. Session Beans, which can be stateless or state-full, provide the functional interface by acting as the controller for the session tier. Message Driven Beans provide an asynchronously listener interface that is used by Java Message Service.

Entity Beans are persistable objects that represent entities in the application; thus, they represent the ORM component provided by the EJB specification. The persistence of Entity Beans can be managed by the developer through the use of bean-managed persistence or through the container through container-managed persistence. If bean-managed persistence is used, the developer must implement all database access code and interact directly with database or interface with another component in order to access the database.

The loosely coupled design principle dictated that applications could integrate EJBs from other applications and even other vendor's applications. EJBs can call other EJBs through the use of arbitrary names, and EJBs themselves can be developed without any prior knowledge of the environment in which they are to be deployed. Overall, the idea is to

provide a very environment and application agnostic architecture for developing and deploying application components.

A key component of the EJB architecture is the services provided to the developer through the EJB container. This includes object persistence through the entity bean API, security and the ability to hook into security APIs, transaction processing, and connection pooling. The container also provides complete component lifecycle management and manages thread behavior. The complete configuration of all of these can be managed through configuration files in a declarative way, or through other graphical user interfaces provided by the container vendor.

Although the EJB specification was adopted initially by many companies, developers soon found out that the EJB specification was fairly complex to develop and deploy. This included a relatively large amount of configuration files that were required to deploy components in an EJB server. Some of these difficulties were not easy to accept without a clear understanding of the benefits the EJB specification was bringing to an application. While developer tools were eventually developed by vendors to aid in EJB development and deployment, many software groups had already begun to switch to alternative APIs and frameworks, including open-source ORMs and application frameworks.

As mentioned above, the EJB specification has contains a persistence layer mechanism for mapping objects, known as Entity Beans, to a relational database. The mapping can be managed by the developer through Bean Managed Persistence (BMP) or by an EJB container through the use of Container Managed Persistence (CMP). In BMP, the developer writes the SQL code, while in CMP, the server develops and executes the SQL calls. In the current EJB release (3.0), this ORM component has been removed from the core EJB specification into its own API, termed Java Persistence API (JPA). The key distinction between JPA and EJB is that JPA does not need an EJB container for deployment, and can be deployed in a basic Java Runtime Environment (JRE), while EJB requires a complete runtime container that has implemented the appropriate EJB interfaces.

There are many EJB commercial and open-source products. Commercial implementations include IBM Websphere, BEA Weblogic, Oracle Application Server, and Macromedia's JRun. Open-source implementations include JBoss, JOnAS, and OpenEJB. The selection between commercial and open-source is usually based on cost and/or the availability of technical support by the vendor. As mentioned above, the EJB architecture attempts provide a number of services to the developer above and beyond an ORM solution. Thus, such a decision to use EJB may be based other services provided by the container or legacy integration components provided by the vendor.

2.3 Java Data Objects

Java Data Objects (JDO) was Sun Microsystem's original specification for Java object persistence prior to issuance of the JPA specification (Sun Microsystems, 2004, 2005). A key difference between JDO and EJB persistence is that objects (i.e., entities) in JDO can be *plain-old Java objects* (POJOS) and are not required to implement specific interfaces. The JDO API is an interface that allows applications to persist these POJOs in a datastore, whereby the datastore itself does not need to be a relational database. In fact the datastore can be a XML, a flat file containing data, or another source of data.

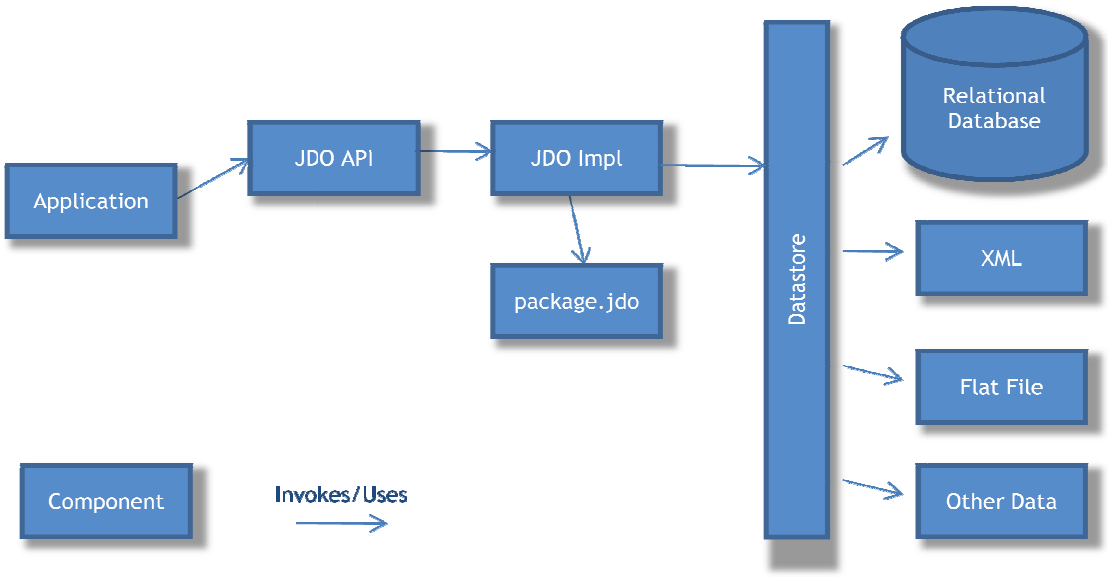

Figure 2.1 JDO architecture overview

The overall JDO architecture consists of the JDO API that is provided by the Sun Specification and an implementation of the specification provided by the vendor or open-source project (Figure 2.1). Since entities, or JDO objects, are POJOs within the JDO implementation, all configurations for mapping objects to the datastore is managed by key configuration file, typically named `package.jdo` by convention.

The primary JDO session object is the `PersistenceManager` that provides an interface to query and manipulate JDO Objects through transactions. The `PersistenceManager` is obtained through a `PersistenceManagerFactory` which is configured through a properties file using name/value settings. If multiple datastores are utilized in an application, then a `PersistenceManagerFactory` must be made available for each.

In JDO, there are actually three types of Java classes: *Persistence Capable*, *Persistence Aware*, and *Normal*. *Persistence Capable* classes can be persisted to a datastore and are enhanced prior to use. *Persistence Aware* classes are utilized to manipulate *Persistence Capable* classes and are generally modified only slightly through the enhancement process. *Normal* classes are not able to be persisted, nor do these classes relate to an any persistable classes.

A JDO implementation, through the use of the `PersistenceManager`, handles the entire lifecycle of a JDO object that has been mapped to an entity in the datastore (Figure 2.2). The transition throughout an object's lifecycle is accomplished through operations made available through the `PersistenceManager`.

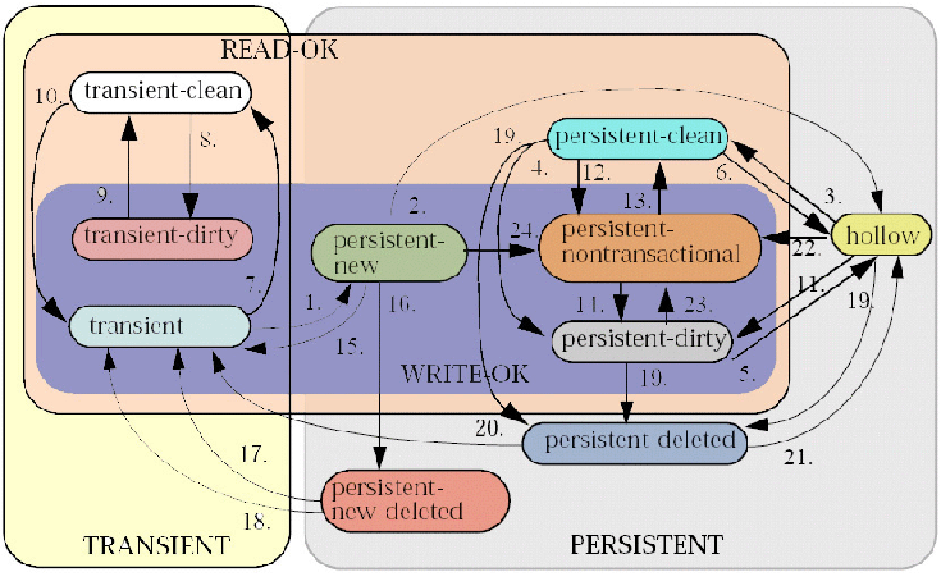

Figure 2.2 JDO lifecycle states (Sun Microsystems, 2005)

The JDO specification calls for JDO implementations provide a query language termed JDOQL, which is an object-oriented query language that provides a way to select objects. With JDO 2.0, an implementation must also provide SQL language support, if the implementations support SQL-based datastores.

2.4 Java Persistence API

As mentioned above, object persistence has been removed from the EJB specification into the JPA specification (Sun Microsystems, 2006a). JPA was introduced in order to reduce the complexity of EJB development issues concerning BMP/CMP in addition to providing a single persistence API for all Java applications. The intent of JPA was to bring together the best ideas from non-standard-based ORMs, such as Hibernate (see below), as well as JDO and commercial vendor products like Oracle's TopLink. Thus, Java would offer a common ORM model that any application could use, be it a server application or a stand-alone application.

Similar to JDO, JPA is based on POJO objects where there are no specific Java classes or interfaces to extend (Figure 2.3). In addition, JPA configuration is both annotation-based, where mappings can be defined within the classes themselves, and XML-based, where mappings are defined in external configurations files. JPA also supports a query language that is similar to SQL and can support static as well as dynamic queries.

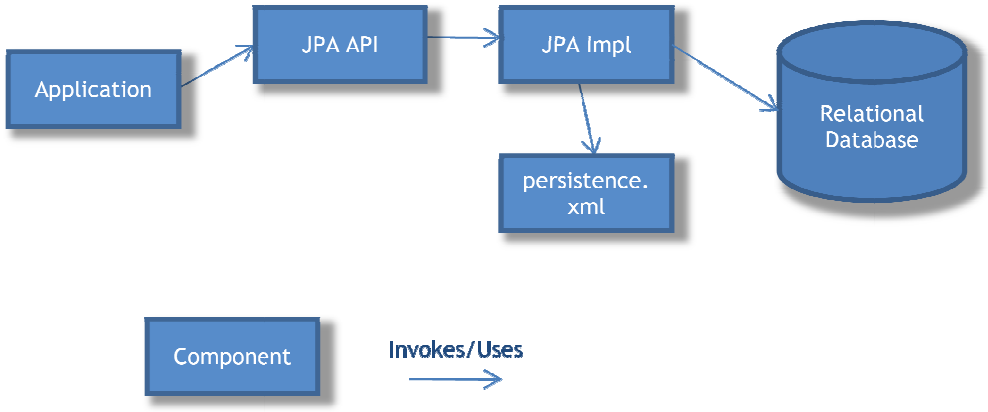

Figure 2.3. JPA architecture overview

Overall, JPA includes the concepts of an `Entity`, the `Entity`'s metadata, an `EntityManager`. An `Entity` is that object that is mapped to the database. Thus, an `Entity` can be made persistent, has a persistent identity, is transactional during creates/updates/deletes, and is not a primitive, primitive wrapper, or a built-in Java object. An `Entity` is described through `Entity` metadata, which can be expressed as annotations within the Java (.java) file or as a separate XML configuration file. When annotations are used, the information is placed directly before the fields or the property accessors in the file.

An `EntityManager` provides the interface on which to perform `Entity` operations, including searching, creating, updating, and deleting an `Entity` or group of Entities. When an `Entity` is obtained through an `EntityManager`, that `Entity` becomes managed by that

`EntityManager`. Thus, the set of managed Entities by an `EntityManager` is called a *persistent context*. Thus, only a single instance of an `Entity`, with a specific `Entity` identity, can exist in a given *persistent context* of an `EntityManager`. An `EntityManager` is configured to manage specific objects, interact with a specific database, and be implemented by a specific JPA provider. The JPA provider is the implementation for the JPA persistent context, which includes the `EntityManager`, the `EntityManager`'s `Query` interface, and the generation of the SQL statements for accessing the database.

An `EntityManager` is obtained through an `EntityManagerFactory`, which is also provided by the implementation. The `EntityManagerFactory` is defined separately in a configuration file (persistence.xml) that is placed in the classpath of the Java Virtual Machine (JVM). Each configuration file may contain multiple `EntityManagerFactory` configurations; thus, multiple *persistence contexts* can be created for a single JVM and be made available to an application.

With the creation of JPA, Sun Microsystems has created a common persistence API for object-relational mapping. They intend to still support both the EJB CMP architecture and JDO; however, their long-term intent is to have applications move to the common JPA.

2.5 Non-Standard-Based ORMs

There are numerous ORMs that are not based on a Sun Microsystems JSR. By far, the most popular is Hibernate, an open-source ORM tool that gained major acceptance since its inception. This is mostly due to Hibernate's ease of use, especially in comparison to the configuration required for EJB entity-relationship mapping. In general, Hibernate operates by holding the mapping between Java domain classes and database tables in external XML configuration files. When the application runs, the Hibernate engine reads the mapping files and dynamically develops the classes that are utilized to manage the object-relational transactions. The Hibernate codebase is managed by JBoss, which itself is a division of Red Hat. Thus, Hibernate is actually managed by a commercial company, although has an open-source developed code base (Hibernate User Documentation, 2007; Pugh and Gradecki, 2004). Hibernate also provides a JPA interface to the core Hibernate ORM code; thus, this framework is also accessible through a standard API.

Another popular Java ORM is Apache Software Foundation's Cayenne, which also includes a complete Graphical User Interface (GUI), known as the *CayenneModeler*. Through the use of the GUI tool, a developer can bind database schemas directly to Java classes. Cayenne supports atomic commits/rollbacks, the generation of SQL code, joins, and sequences. Cayenne also includes object caching, an object query language, pre-

fetching relationships, object-inheritance, and database auto-detection (Cayenne User Documentation, 2007).

One of the first open-source Java ORMs is the Castor Project. Castor provides a Java persistence framework as well as a Java object model for serializing objects to XML and de-serializing XML back to objects. Castor provides in-memory caching and write-at-commits in order to reduce JDBC access calls. Castor also provides two-phase commit transactions, rollbacks, and deadlock detection (Castor User Documentation, 2007).

An abundance of other ORMs have been developed and are usually available as open-source. All of these generally work on the idea of using Java POJOs and the use of configuration files in order to map Java classes to RDMS tables. These other ORMs include, but are not limited to the following: Carbonado (SourceForge Project); Torque (Apache Software Foundation); Hydrate (SourceForge Project); Ibatis (Apache Software Foundation); SimpleORM; and, JDBCPersistence.

2.6 ORM Object Caching

The caching of retrieved objects is a key component of ORM solutions in order to improve performance. Overall, caching is intended to reduce the number of database access calls by saving data that has already been retrieved from the database. The cache storage can be managed based on

updates/deletes to the persistence model, and can be periodically emptied in order to keep fresh data in the cache.

Caching generally occurs in two levels, a first-level cache and a second-level cache. First level caches work directly within a single JVM, while second-level caches can work within a single JVM or can be distributed across JVMs. In addition, query caches can be implemented that maintain the results of queries to the database.

Caching usually allows a way to configure specific objects with a caching strategy, including options for reading and writing/updating data. A read-only cache is utilized for objects that are usually read from the database, but not updated. A read/write cache is used for objects that are both read and updated. A read/write cache adds additional computational overhead, thus it is not as fast as a read-only cache. A nonstrict read/write cache is utilized when data is read and updated, but updated only rarely. Thus, the cache does not guarantee to keep two separate threads from modifying the data at the same time.

Caching can be a key element to the performance of an ORM, with several commercial ORMs providing distributed cache components. For example, the commercial JPA/JDO Kodo provides such a distributed cache management (Figure 2.4). Using a cache architecture reduces read/writes

across several JVMs, an even several nodes while allowing all nodes to remain consistent with respect to the current state of the database.

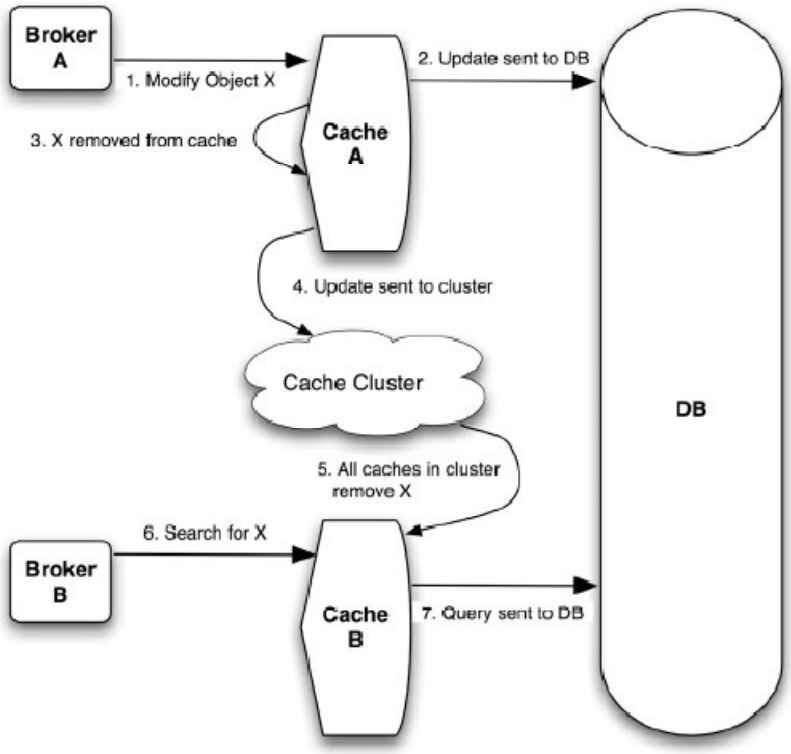

Figure 2.4 Distributed cache during database update (Linskey and Prud'hommeaux, 2007)

2.7 Summary

Several commercial and open-source Java ORMs are currently available that are based on Sun Microsystems standards and non-standard APIs. The EJB ORM component termed Entity Beans has been criticized for being too difficult and complex, and thus has given way to more manageable frameworks such as Hibernate. JDO and JPA offer alternatives to EJB entity

beans and allow the use of POJOs. The current Sun Microsystem strategy is to move all data persistent to relatively new JPA architecture, while still supporting the EJB Entity Bean and JDO standards. JDO and JPA are similar with respect to their APIs and the management of objects, including their POJO nature and the absence of a required runtime container.

Open-source frameworks have also been very popular, with Hibernate being the most widespread in the industry. Hibernate also provides a JPA interface to its core ORM engine; thus, Hibernate users have the option of using the JPA interface with an underlying Hibernate ORM engine. There are also an abundance of other open-source solutions that have varying degrees of industry support and an associated developer community.

3 Assessing ORM Performance

3.1 Overview

Extensive research exists on the benchmarking of RDBMs object-oriented database management systems (ODBMS), with a majority of the studies being performed in the 1980's and 1990's when these technologies were beginning to be used extensively in industry. While RDMSs and ODBMs performance studies exist, the literature is somewhat bare with respect to evaluating the relative performance of Java ORMs, as only a handful of papers examine the issue in an attempt to discern any differences between ORM design decisions. In addition, while other studies are present that examine Java middleware performance, these studies have been focused on addressing CORBA (Harkema et. al., 2002), CORBA and RMI (Juric et. al., 2006), and web services (Juric et. al., 2000).

3.2 Relational Database Performance

The literature includes numerous studies into database performance, with the idea of performance measurements going back over twenty years (Demurjian et al, 1985). Many of these studies involve investigating how benchmarks are constructed. Most of the database vendors include private benchmarking tests that are also run. While many benchmarks have been

developed and implemented, the industry has generally settled upon the use of a few.

The first performance benchmark was called the Wisconsin benchmark (Bitton et al., 1987). This benchmark consisted of a set of 32 retrieval and update commands in addition to a script for mult-user executions. The benchmark provides two performance measures: execution time and the throughput of the system while running sixteen scripts at the same time. The benchmark does not include a cost measure.

Another benchmark utilized simple queries for the performance measure (Rubenstein et al., 1987). These researchers proposed a number of factors that improve performance, including caching the entire database, how to avoid the overhead associated with query optimization, and the use of physical links for pre-joining.

The benchmarking of an isolated database system is challenged in research that proposed the benchmarking of database applications (Doppelhammer et al., 1997). These researchers state that isolated benchmarks do not represent real-world scenarios where applications are the interface to the database. A TPC-D benchmark is proposed with results provided for the SAP R/3 system, which is an integrate business system. A similar idea concerning the testing of database applications has also been proposed (Chays et al, 2000).

A database benchmark (TPC-W) specifically directed toward web applications was developed by the TPC (Poess and Floyd, 2000). This benchmark was developed due an industry need for measure for both software and hardware for an ecommerce application. The benchmark is based on an online shopping scenario and includes a combination of static web content and dynamic web content. Additional work regarding web applications and results utilizing the TPC-W was performed by Deng et al., 2004.

Other researchers have also challenged the use of standard industry benchmarks (Seng et al., 2005). These researchers claim standard benchmarks are domain-specific and that results are just estimates of potential system performance for pre-defined applications. Instead, the researchers propose a benchmark that is both computer-assisted and is based on user requirements.

Researchers have measured the performance differences between conventional computer architectures on which database applications run and computer architectures that have been developed specifically for databases (Yao, et al., 1987). Results from the study indicate that database machine architectures are faster performers when compared to traditional computer architectures.

3.3 Object Database Performance

The key strength of using an ODBMS is to gain a more flexible and maintainable system for complex data models, while improving database performance in such a complex system. As an application generally grows in size, such a performance improvement can be realized more in an ODBMS unless the relational-database system is continually tuned to match application requirements. Meanwhile ODBMSs do not provide an easy way to provide ad-hoc queries. Thus, ODBMS are better for applications that have a complex hierarchy of classes such as engineering applications, while RDMSs are generally better for information systems where data models are relatively simple.

The benchmarking of ODBMSs occurred mostly during the late 1980's and 1990's when the industry began questioning the performance these systems (Zyl et. al., 2006). Several of the ODMBS benchmarks were developed include the *001 Benchmark* (Cattell and Skeen, 1992), the *007 Benchmark* (Carey et al. 1993), and the *Hypermodel* (Anderson et. al., 1998). One of the most popular is the *007 Benchmark* which is based on the idea of a design library that is composed of parts and assemblies, an approach that is similar to a number of object databases such as CAD, CAM, and CASE. The *007 Benchmark* is generally used to test database associative operations, updates to indexed attributes, and traversals. The benchmark

suite is relatively large, and results are available for several major object database vendor products (Carey et al. 1993).

3.4 Java ORM Performance

An attempt at applying the *OO7 Benchmark* to Java ORMs was performed by Sun Microsystems (Jordan, 2004). In fact, this study is by far the most complete overall evaluation of Java ORMs. In the study, a number of ORMs are compared, including EJB-CMP, EJB-BMP, JDBC, and JDO. The study was primarily aimed at setting up the framework for evaluating ORMs, although the study indicated that JDO was the best performer when compared to the other ORMs. The study did not provide any analysis of the results or how various performance strategies factored into the results.

Another use of the *007 Benchmark* and Java ORMs is presented in Zyle, et. al., 2006. These researchers actually cite the use of the *007 Benchmark* in an ORM study was taken from the Jordan 2004 study. In the Zyle, et. al, study, the *007 Benchmark* tests were use to evaluate the performance aspects of an object database, db4o, versus an ORM product, Hibernate. Overall, the object database had better performance when compared to the ORM. The study did not compare Java ORMs against other Java ORMs, nor did it investigate EJB-CMP, JDO, or JPA persistence mechanisms.

While the Jordan and then Zyle et. al. utilized the *007 Benchmark*, another approach is presented in a study originally developed by The Middleware Company (Martin, 2005). In this study, *The Testbed of Object Relational Products for Enterprise Distributed Objects* (TORPEDO), provides a set of application-level operations that are used to evaluate whether database access optimizations have been implemented in an ORM. No results for TORPEDO testing are provided in the literature, nor does TORPEDO compare JPA against the other ORM frameworks. However, the paper mentions that TORPEDO can be used for evaluating Hibernate, EJB-CMP, JDO, and Oracle's TopLink.

One unpublished Java ORM benchmark, *Pole Position*, provides an automated test suite for examining object databases as well as ORM performance using a series of database access calls (Pole Position, 2007). Documentation provided on the Pole Position website provides some high-level results comparing Hibernate and JDO against hand-coded JDBC calls. These results indicate that the use of an ORM has a negative impact on an application's performance compared to hand-coded JDBC, although the use of caching can dramatically improve performance. In addition, the use of an object database had better performance when compared to the use of an ORM. The Pole Position documentation does not provide any evaluation of the results, nor is there an evaluation of JPA or more than a single JDO implementation.

3.5 Summary

The literature contains only two studies that provide any evaluation of performance between various Java ORM designs and implementations. An additional study (Martin, 2005) compared a Java ORM against an object database; however, no comparison between Java ORMs was made.

While an open-source benchmark, Pole Position, is available to run tests between various Java ORMs, no discussion of the results is provided and no analysis is attempted to evaluate how ORM designs factored into the performance differences. Finally, no study or benchmark attempts to compare multiple JPA or JDO implementations, nor provides an analysis of how various design decisions affect performance.

4 A Performance Evaluation Framework

4.1 Overview

The intent of the current research was to develop an objective framework for evaluating ORM performance and provide an implementation of that framework for a selected number of Java ORMs. The implementation of the framework was intended to fill in gaps in the literature concerning Java ORM performance evaluations. Most notably, the literature does not contain an evaluation of JPA and JDO, including the Hibernate implementation of JPA. In addition, there is no evaluation of multiple JPA and/or JDO implementations. Thus, this research evaluated the performance of two JPA implementations and two JDO implementations with the intent of comparing the performance of JPA with JDO in addition to comparing JPA implementations and JDO implementations.

The evaluation of the ORMs was performed by defining the performance test cases of the framework, building a benchmark application data model, and then creating a database schema to represent the persistent storage of that model. Each selected ORM was then used to develop domain classes and the necessary database access code to map the database entities and their associated relationships to these domain classes. After the domain classes and database access code were developed, the

suites of performance test cases were executed to evaluate the performance of each ORM.

4.2 Performance Test Cases

While an application's persistent data model can be accessed in a nearly unlimited number of scenarios, this research proposes a suite of common data model access use cases under which the majority of database access occurs. Thus, while the test cases do not cover all database usage scenarios, the test cases *can* be used to determine the relative performance of an ORM with regard to processing times, database access, and the use of object caching within the ORM implementation. Specifically, eight performance test cases (Table 4.1) were developed in an attempt to provide a snapshot of the relative performance of the ORMs across a range of use cases that would occur within an application:

- `LoadObjectTwice`
- `LoadObjectTwiceBtxTrx`
- `LoadPartialObject`
- `LazyLoadObjectGraph`
- `EagerLoadObjects`
- `BulkLoadObjects`
- `BulkObjectSave`
- `LoadAndSave`

The test cases, based on ideas identified in the TORPEDO framework (Martin, 2005), attempted to illuminate the ability or inability of the ORM to

perform efficient database access calls and internally manage objects in order to keep remote database access calls to a minimum.

Table 4.1 Performance test cases

Test Case	Description	Goal of the Test
`LoadObjectTwice`	The same object is loaded twice within the same transaction.	Determine time required to retrieve same object within the same transaction, and whether multiple database access calls are made.
`LoadObjectTwiceBtxTrx`	The same object is loaded twice within the same thread, but inside two separate transactions.	Determine time required to retrieve the same object between transactions, and whether multiple database access calls are made.
`LoadPartialObject`	A group of objects are partially loaded with only selected fields populated during the transaction.	Determine whether the ORM can retrieve a partially-loaded object, the time required to perform this task, and the number of database access calls.
`LazyLoadObjectGraph`	A complex object graph is lazy loaded on an as-needed basis during the transaction.	Determine time required to lazy-load an object graph, including the number of database access calls.
`EagerLoadObjects`	A complex object graph is eagerly-loaded with all fields populated during the transaction.	Determine time required to eagerly-load an object graph, including the number of database access calls
`BulkLoadObjects`	Complete sets of related objects are loaded within the same transaction.	Determine time required to bulk-load related objects including the number of database access calls.
`BulkObjectSave`	A large set of objects are saved within the same transaction.	Determine time required to bulk save objects within the same transaction, including the number of database access calls.
`LoadAndSave`	A set of database reads is completed in order to create an object graph and then save it to the database.	Determine time required to perform multiple reads and execute complex object graph save within the same transaction, including the number of database calls.

4.3 Application Domain Model

In order to provide a runtime implementation of the test cases, an application domain model was developed. The selected domain was an airline reservation system that would allow airline passengers to book airline flights and pay for tickets (Figure 4.1). The model is fairly self-explanatory: passengers can purchase tickets which can be associated with specific flights. Tickets can also be associated with payments, which in turn have a payment status of paid, not paid, and pending.

The data model contained common entity-relationship multiplicities found in all databases, including one-to-one, one-to-many, and the use of a join table to break a many-to-many relationship into two, one-to-many relationships. In addition, the entities themselves contained a mix of data types, including characters, numeric types, and dates. Overall, the airline reservation model was selected as it represents an intuitive model that can be easily visualized, in addition to maintaining entity-relationships found in common database schemas. It should also be noted that a real-life domain was selected instead of a generic object domain in order to provide a more instinctive exploration of the test cases.

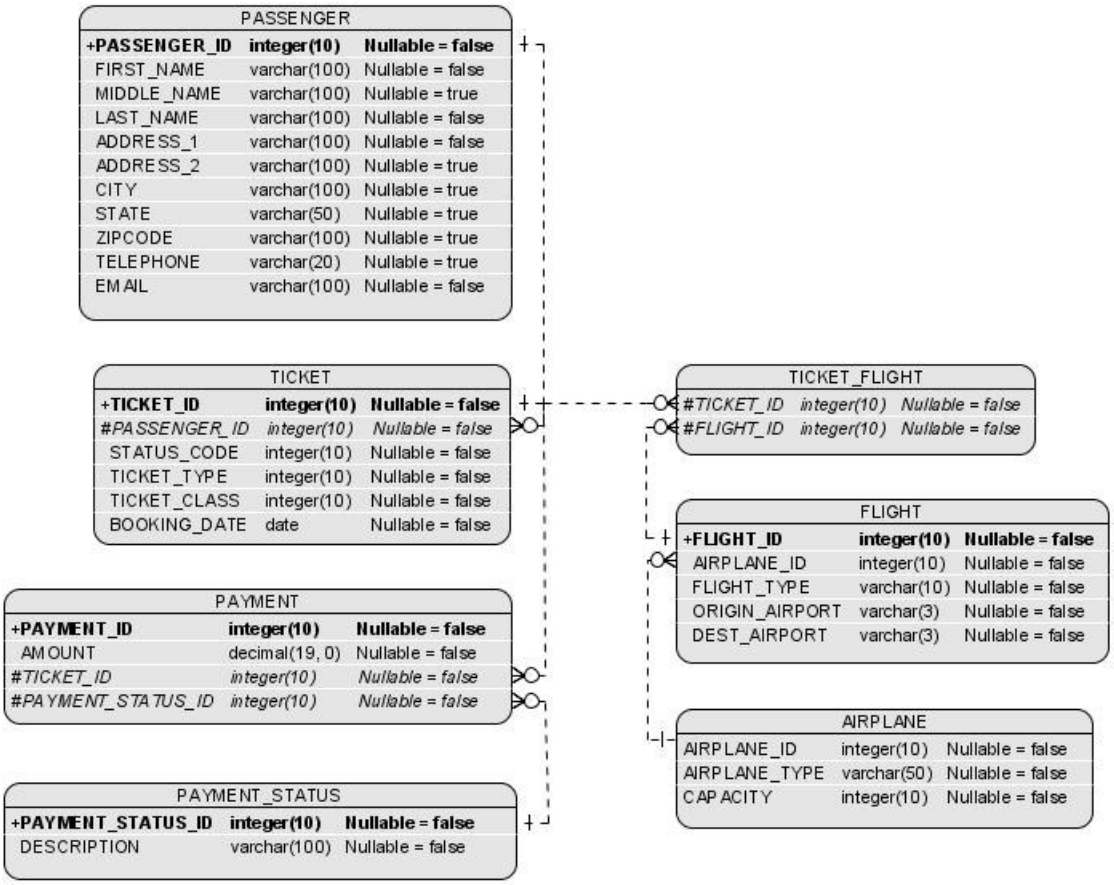

Figure 4.1 Airline reservation system entity-relationship model

The airline reservation ER model was used to develop a persistent Java class model (Figure 4.2). This Java model represented the application's view of the ER data model, and included all persistent entities identified in the ER model (Figure 4.1). The intent of developing the common Java model was to provide consistency between ORMs with respect to the class relationships and data types assigned to individual fields within each class.

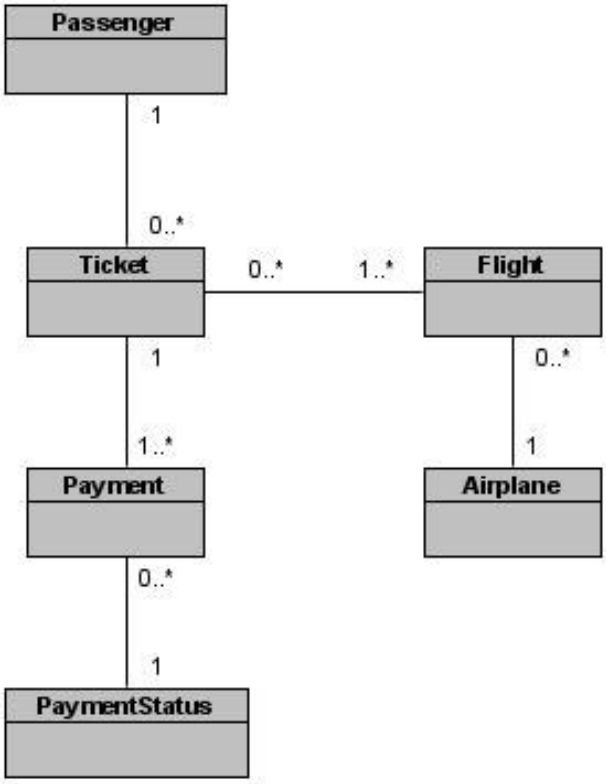

Figure 4.2 Airline reservation class diagram

4.4 Selected ORMs

Implementations of the performance test cases using the application domain model were undertaken for a selected number of Java ORMs. Hibernate's JPA implementation and OpenJPA were the JPA implementations that were selected (Table 4.2). The Hibernate open-source implementation was selected due to its widespread use in the industry. OpenJPA, which is also open-source, was selected due to its widespread use in the industry and its use as the ORM engine with the commercial Kodo JPA/JDO implementation.

The JDO implementations selected were JPOX and Speedo (Table 4.2). JPOX was chosen as it is the reference implementation of the JDO 2.0 specification and is open-source. Speedo was selected as it is a popular open-source implementation of JDO 2.0 maintained by ObjectWeb.

Table 4.2 ORMs evaluated

ORM	Version	Specification	Company	Type
Hibernate	3.3	JPA	JBoss(Red Hat)	Open-source
OpenJPA	1.0	JPA	Apache	Open-source
JPOX	1.2	JDO	JPOX	Open-source
Speedo	4.1	JDO	ObjectWeb	Open-source

4.5 `TestIF`

A test interface (`TestIF`), as identified in Figure 4.3, was developed to represent each of the eight performance test cases (Table 4.1). The `TestIF` represented the common interface that was executed from the testing harness during the performance evaluation.

An ORM implementation of the `TestIF` was then developed for each of the selected ORMs (Figure 4.3). Thus, each test case was translated to the domain model in order to develop a domain model use case that fulfilled the goal of the test, as identified in Table 4.1. The ORM implementations were developed as described below in Section 4.7.

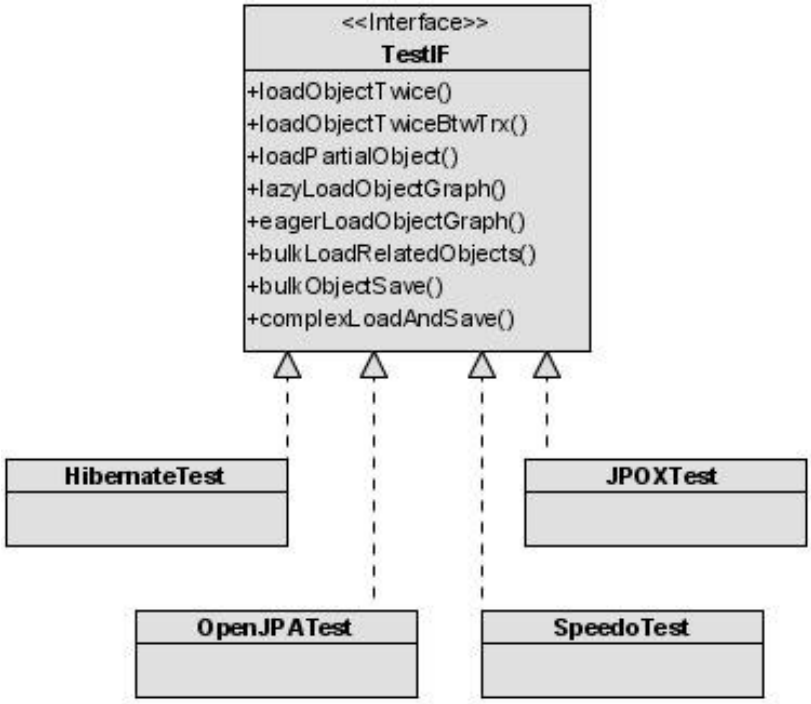

Figure 4.3 Performance testing interface class diagram

The actual steps to complete each test case were duplicated within each `TestIF` implementation (Table 4.3). In this way, each `TestIF` implementation was required to complete the same number of steps while leaving the implementation of those steps to the interface provided by the ORM. For example, each ORM was required to execute all test cases within a transaction.

Again, the intent of the research was to provide a consistent approach to the database access scenarios, thus, the identification of the test case mapping steps was critical to making sure each ORM implementation executed the same number of steps.

Table 4.3 Implementation steps for performance test cases

Test Case	Airline Reservation Implementation
`LoadObjectTwice`	1.) Start Transaction 2.) Retrieve `Passenger` object 3.) Retrieve `Passenger` object again 4.) Close Transaction
`LoadObjectTwiceBtwTrx`	1.) Start Transaction 2.) Retrieve `Passenger` object 3.) Close Transaction 4.) Start Transaction 5.) Retrieve `Passenger` object again 6.) Close Transaction
`LoadPartialObject`	1.) Start Transaction 2.) Retrieve `Passenger` object loaded with `passengerId`, `phone`, `email` 3.) Close Transaction
`LazyLoadObjects`	1.) Start Transaction 2.) Retrieve `Passenger` object 3.) Retrieve `Passenger` object's `Ticket`s 4.) Retrieve each `Ticket`'s `Flight` objects 5.) Retrieve each `Ticket`'s `Payment` objects 6.) Close Transaction
`EagerLoadObjects`	1.) Start Transaction 2.) Load List of `Payment` objects with associated `PaymentStatus` objects 3.) Close Transaction
`BulkLoadObjects`	1.) Start Transaction 2.) Retrieve List of `Ticket` objects 3.) Bulk retrieve each `Ticket`'s Flights 4.) Bulk retrieve each `Ticket`'s Payments
`BulkObjectSave`	1.) Start Transaction 2.) Create 10 `Passenger` objects 3.) Save 10 `Passenger` objects 4.) Close Transaction
`LoadAndSave`	1.) Start Transaction 2.) Retrieve `Passenger` 3.) Create `Ticket` 4.) Retrieve `Flight` and add to `Ticket` 5.) Create `Payment` 6.) Retrieve `PaymentStatus` and add to `Payment` 7.) Add `Payment` to `Ticket` 8.) Persist `Ticket` 9.) Close Transaction

4.6 Performance Metrics

The performance of each ORM to implement each test case was measured in terms of time, in milliseconds, and the number of database access calls required to complete the test. The time was collected for a single execution of each test case and for a collection of 100 executions of each test case for each ORM solution (Table 4.4). Thus, the performance was measured for the initial execution of the test case as well as the ability of the ORM to cache data for subsequent calls to the test case within the same thread. For each of these scenarios (single and 100 executions), a set of 10 runs were performed with a mean time being recorded for the test case for the respective ORM.

Table 4.4 Performance metrics

Metric	Units
Time to execute test case 1 time	ms
Time to execute test case 100 times	ms
Number of database access calls to execute test case 1 time	NA

In addition to the multiple iteration comparison to assess caching, a selected evaluation of the Hibernate JPA implementation with and without caching was undertaken. This evaluation was performed to determine the overall affect of caching on reducing the time required to process 100 iterations of the loading test cases (`LazyLoadObjects`, `EagerLoadObjects`, `BulkLoadObjects`)

4.7 Software and Hardware Test Bed

Each ORM implementation was developed using the Java 6 SDK, and the test case execution was performed using the Java 6 JRE. The Eclipse integrated development environment (IDE) was utilized to develop each ORM implementation of the `TestIF`, and the Ant build tool was used to build and execute each ORM performance test suite.

The database utilized for the development and testing was Oracle 10G Personal Edition. Data load scripts were created and run to load the Oracle database schema with sample data for the performance tests.

The application, including each ORM implementation, in addition to the local Oracle database was tested on Fedora Core 6 installed on a Dell Inspiron Notebook. The Dell contained an Intel Core 2 Duo processor with 2 GB of RAM.

The collection of the performance metrics was assisted through the use of P6Spy and JProfiler. P6Spy allowed the collection of the SQL calls that were made to the database, and JProfiler allowed an examination of the actual call stack. In addition, a custom Logger class was constructed to measure the time to execute each test case for each `TestIF` ORM implementation.

5 ORM Performance Results

5.1 Overview

The complete results for the performance test cases are provided within the following chapter. The results below are divided into the single execution results and the multiple execution results. Also, included are figures for a selected number of ORM comparisons between the various performance test cases. A discussion of these results is provided in Chapter 6.

5.2 Single Execution of Performance Test Cases

The single execution test results (Table 5.1, Figure 5.1) indicated the Hibernate JPA implementation was the quickest performer overall, as it held the shortest times or tied for the shortest times for all tests. The OpenJPA implementation was the second fastest performer for all tests, followed by the JDO implementations JPOX and Speedo. Thus, overall, the JPA implementations performed better when compared to the JDO implementations.

Hibernate was noticeably faster on the `LazyLoadObjects`, `BulkObjectSave`, and the `LoadAndSave` test cases (Figure 5.2). Regarding lazy loading, Hibernate was at least two times faster, when compared to OpenJPA, and up to three times faster, when compared to the

JDO ORMs. Hibernate's `BulkObjectSave` test case was approximately four times faster than the next faster ORM (OpenJPA). Hibernate's `LoadAndSave` was also relatively fast as it was at least two times faster than the next fastest, which was also OpenJPA.

The OpenJPA and JPOX ORMs exhibited similar times for nearly all tests, with only the `LoadPartialObject` test case showing any real difference between these two ORMs. Thus, the OpenJPA and JPOX ORMs were the closest performers for the single execution test cases.

Speedo was generally the slowest of the ORMs in the single execution, as it took approximately twice as long as any other ORM on the `BulkLoadObjects` and `LoadAndSave` test cases. Speedo was also noticeably slower on the `LoadPartialObject` test case. However, Speedo showed similar performance on `LoadObjectTwice` and `LoadObjectTwiceBtwTrx` test cases.

Table 5.1 Time (ms) to complete single execution of test cases

Test Case	Hibernate	OpenJPA	JPOX	Speedo
`LoadObjectTwice`	30	60	30	30
`LoadObjectTwiceBtwTrx`	30	60	30	30
`LoadPartialObject`	30	30	90	210
`LazyLoadObjects`	40	120	100	80
`EagerLoadObjects`	50	70	90	110
`BulkLoadObjects`	60	100	130	290
`BulkObjectSave`	60	230	280	300
`LoadAndSave`	70	170	180	380

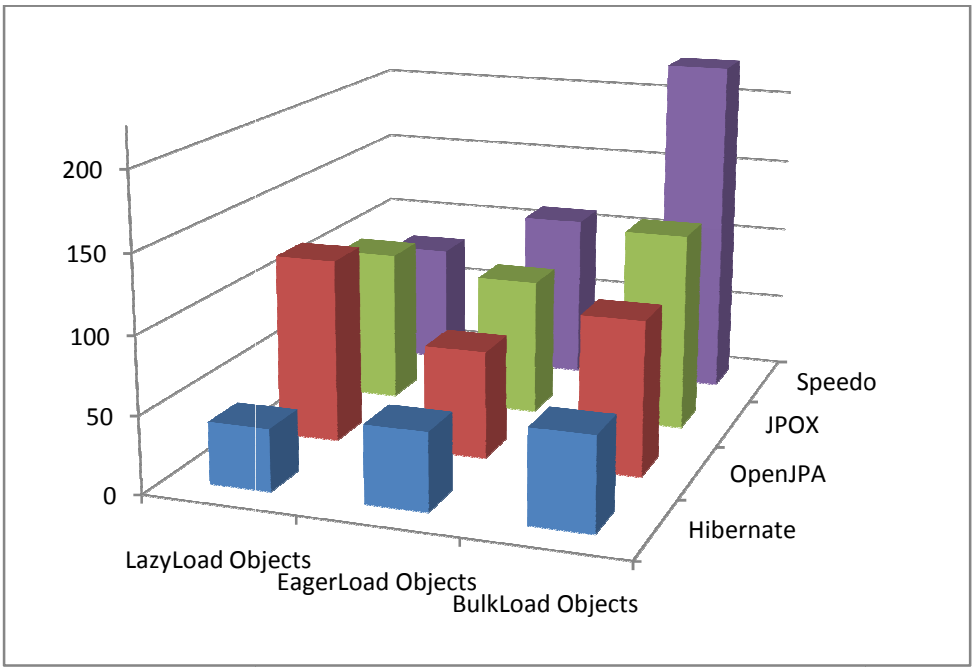

Figure 5.1 Time (ms) to complete single execution of loading test cases

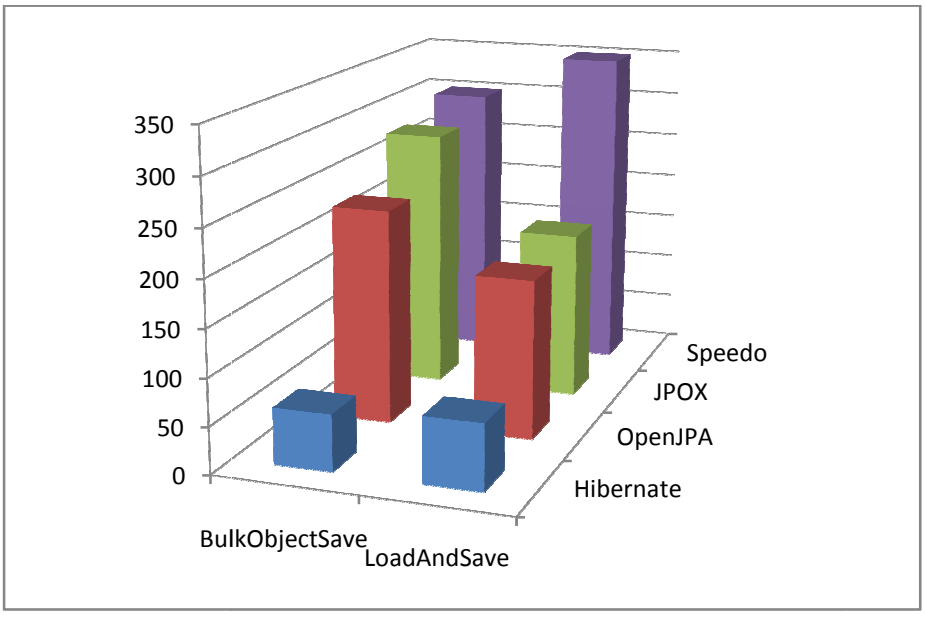

Figure 5.2 Time (ms) to complete single execution of `BulkObjectSave` and `LoadAndSave` test cases

The number of database calls to execute each test case (Table 5.2) was identical between the JPA implementations with the exception of the `BulkObjectSave` and `LoadAndSave` test cases. For these test cases, Hibernate was able to batch the insert calls while OpenJPA was not.

JPOX and Speedo both had more database calls compared to the JPA implementations. Both JDOs had four more accesses for the `EagerLoadObjects` and double the database calls for the `BulkObjectSave`.

Speedo maintained the overall highest number of database calls for all test cases. Most notably, Speedo contained five more calls for the `LoadPartialObject` and `LazyLoadObjects`.

Table 5.2 Number of database calls for single execution of test cases

Test Case	Hibernate	OpenJPA	JPOX	Speedo
`LoadObjectTwice`	1	1	1	1
`LoadObjectTwiceBtwTrx`	1	1	1	1
`LoadPartialObject`	1	1	1	5
`LazyLoadObjects`	4	4	4	9
`EagerLoadObjects`	1	1	5	5
`BulkLoadObjects`	2	2	3	5
`BulkObjectSave`	11	20	20	20
`LoadAndSave`	8	8	10	10

5.3 Multiple Executions of Performance Test Cases

The results of the multiple (100) executions of the test cases indicate the ORMs are more similar once the objects are loaded into memory (Table 5.3, Figure 5.3). Thus, a significant about of the variance can be attributed to the differences in the initial load of the objects during the first execution of the test case in the thread.

The overall internal caching abilities of JPOX and Speedo are somewhat slower when compared to the Hibernate plugin and OpenJPA. Noticeable differences are observed for the `LoadPartialObject` and `LazyLoadObject` test cases between the JPA and JDO implementations collectively. Also noticeable differences are noted for `LoadAndSave` test case where Speedo was over 50 percent slower when compared to the other ORMs.

Table 5.3 Time (ms) to complete 100 executions of test cases

Test Case	Hibernate	OpenJPA	JPOX	Speedo
`LoadObjectTwice`	70	100	180	100
`LoadObjectTwiceBtwTrx`	100	110	180	100
`LoadPartialObject`	90	70	300	350
`LazyLoadObjects`	150	260	450	490
`EagerLoadObjects`	230	290	320	350
`BulkLoadObjects`	250	250	280	390
`BulkObjectSave`	1680	1790	2100	2140
`LoadAndSave`	730	720	810	1250

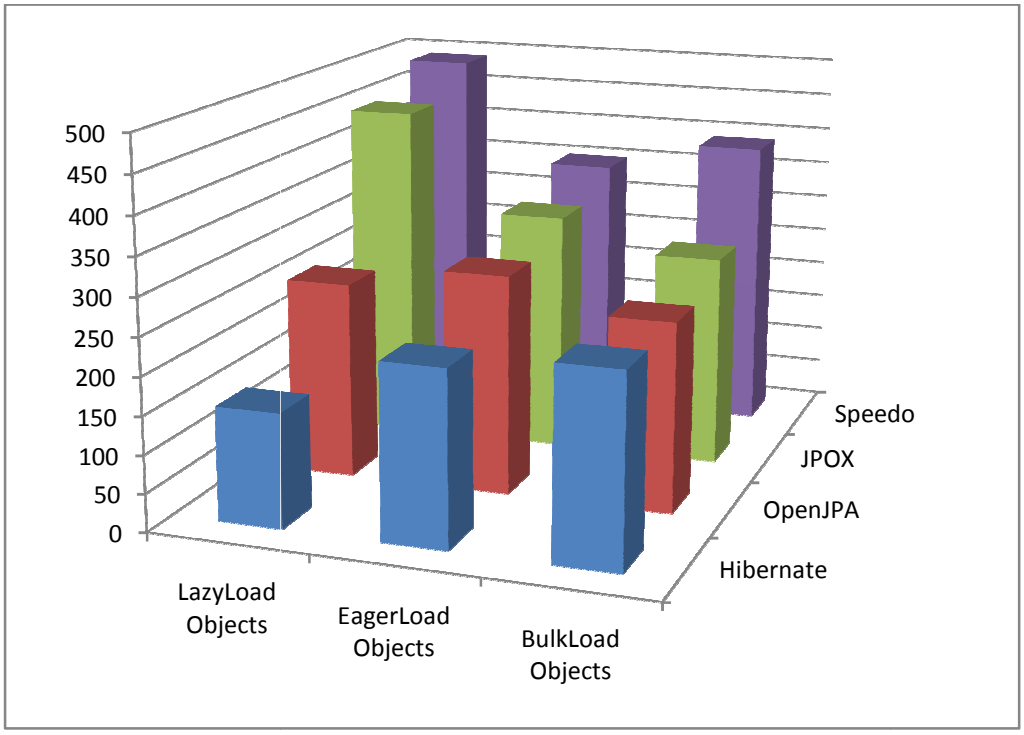

Figure 5.3 Time (ms) to execute 100 executions of selected test cases

5.4 Cache vs. No Cache

The results for the Hibernate object loading test cases under cache and no cache configurations indicated dramatic differences when caching is enabled for multiple requests (Table 5.4, Figure 5.4). When caching was enabled, `LazyLoadObjects` experience almost 300% gain in performance, while `EagerLoadObjects` experienced 70% gain. `BulkLoadObjects` exhibited a 140% increase in performance with caching enabled. Overall, caching had a dramatic affect on improving test case performance when multiple iterations of the test case were performed.

Table 5.4 Time (ms) to execute 100 executions of loading test cases for Hibernate under cache and no cache configurations.

Test Case	Cache	No Cache	Performance Gain (%)
`LazyLoadObjects`	150	590	293
`EagerLoadObjects`	230	390	70
`BulkLoadObjects`	250	600	140

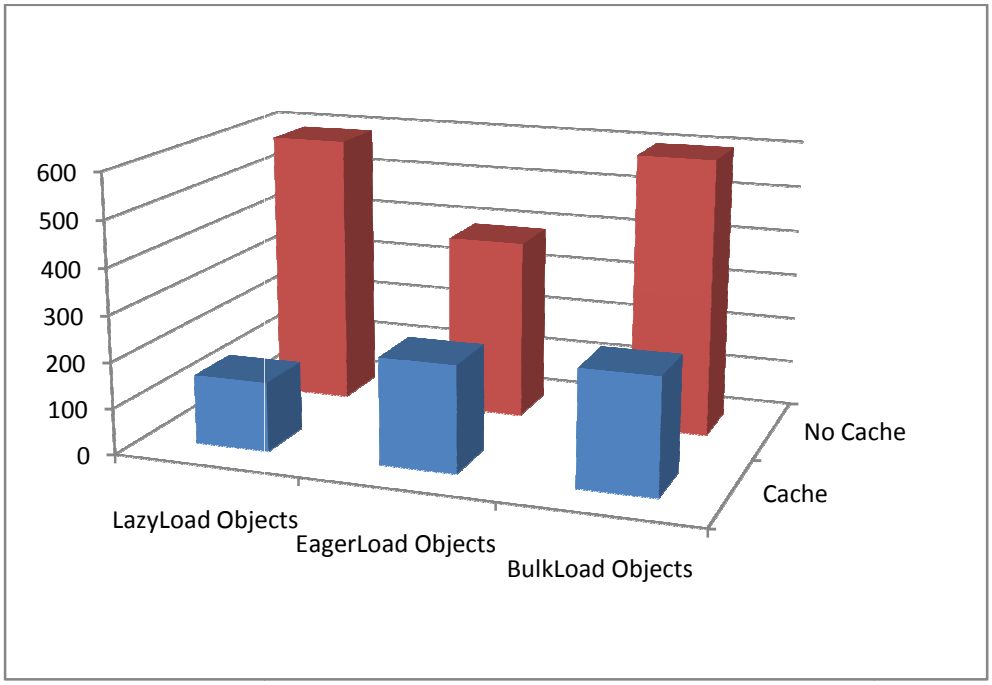

Figure 5.4 Time (ms) to execute 100 executions of loading test cases for Hibernate under cache and no cache configurations.

6 Analysis and Discussion of Results

6.1 Overall Performance

On the single executions, the JPA implementations were better performers when compared to the JDO implementations, with Hibernate being the overall best performer of all implementations. Of the two JDO implementations, JPOX was the better performer, and Speedo was the overall slowest of all the implementations. In addition, JPOX was generally close to OpenJPA for nearly all test cases and even performed better in `LazyLoadObjects`.

While the relative processing times for all test cases illustrate there are performance differences between the various ORMs, these variances may not necessarily translate into absolute performance differences. For example, the complete `LazyLoadObjects` range of values from the slowest (120 ms) to the fastest (40 ms) was just 80 ms. In this case, a difference of 80 ms may end up as a significant performance difference in an method that has a running time of 100 ms, but may not be a significant performance consideration in a method that executes in 5000 ms. Thus, the context of the application would be an important determinant as to whether the choice of the ORM would, in fact, had an impact on meeting performance requirements for that application. For example, a relative

variance between the slowest and the fastest for `LazyLoadObjects`, which is a three-fold difference, may be a performance problem if an application can load these objects in, say, 10 seconds (for Hibernate) instead of 30 seconds (for OpenJPA).

The use of caching tended to level out the performance differences between the various ORMs. Once the objects were retrieved from the database and held in memory, the speed of the ORM's caching mechanism were generally equal.

6.2 LoadObjectTwice

All implementations were equal with respect to this test case, with the exception of OpenJPA which performed relatively slower. A review of the processing time for OpenJPA indicated this difference was related to initializing the cache for the single execution, as OpenJPA was actually a relative fast performer for this test case during the multiple executions.

With regard to database access calls, all ORMs were able to execute this test case by accessing the database only once. Thus, the subsequent call to retrieve the object the second time was made to the local object cache. The cache implementations were relatively similar for the multiple executions, with the exception of the JPOX cache which was noticeably slower.

In summary, the results of the test case indicated the ORMs performed equally in their respective database access strategy for loading the same object twice within the same transaction. The differences in processing time were related to cache initialization and cache performance.

6.3 LoadObjectTwiceBtwTrx

The results of this test case were identical to the results for the `LoadObjectTwice` for the single executions. Again, OpenJPA exhibited a slower time, which was related to initializing the cache for the single execution.

With regard to database access, all ORMs were able to execute the test case with only one database access call. Thus, all ORMs were able to open multiple transactions and find the same object within the local cache without multiple database calls. Also similar to `LoadObjectTwice`, the cache implementations were the same for the multiple executions, with the exception of the JPOX cache which was noticeably slower.

The similarities between the processing times for the `LoadObjectTwice` test and the `LoadObjectTwiceBtwTrx` test are most likely attributed to the resolution of the system clock times returned by the test harness during the test runs. Thus, while both of these test cases actually loaded the same object twice, the

`LoadObjectTwiceBtwTrx` performed more actions as it had to also start and stop two transactions while `LoadObjectTwice` only had to start/stop a single transaction. Thus, the overall unit of work was greater for `LoadObjectTwiceBtwTrx` although the times were identical due to the resolution of the system clock in measuring the processing times for the test cases.

In summary, the results of the test case indicate the ORMs performed equally in their respective database access strategy for loading the same object twice between transactions. The differences in processing time were related to cache initialization and cache performance.

6.4 LoadPartialObject

This test case attempted retrieved contact information for a `Passenger` object without returning the entire object. Thus, the test case was to return just the `Passenger.passengerId`, `Passenger.phone`, and `Passenger.email` without returning other data in the Passenger object. In addition, the query for loading the partial object was looped five times for each ORM for this test case.

The Hibernate implementation allowed for the most elegant implementation for this test case by allowing a simple JPA `NamedQuery` to be added as an annotation in the `Passenger` class. This query could then

be called by name through the ORM's interface:

```
@NamedQuery(name="getContactInfo",
query="SELECT new Passenger(passengerId, phone,
email) from Passenger p where p.passengerId = ?")
```

OpenJPA provided a similar solution with the ability to insert a `NamedQuery` as an annotation; however, the results of the query were returned in an object array in the OpenJPA implementation, while the Hibernate implementation returned an actual `Passenger` object.

The JDO implementations did not provide a means to partially load an object's data either through annotations or directly through the JDO API. As a result, an entire `Passenger` object was required to be retrieved with a normal query in order to complete the test case for the JDO implementations.

The key difference between this test case and the previous test cases discussed (`LoadObjectTwice`, `LoadObjectTwiceBtwTrx`) is that `LoadPartialObject` was executed as a query in the ORM implementation while the other test cases were executed as object lookups by object identifiers. For example, in Speedo, `LoadPartialObject` test case implementation was executed as the following:

```
Query query = pm.newQuery(Passenger.class);
query.declareParameters("Long id");
query.setFilter("id == passengerId");
Passenger pass = (Passenger)((List)query.execute(
          new Long(10))).get(0);
```

While the `LoadObjectTwice` and `LoadObjectTwiceBtwTrx` test case implementations were executed as object queries based on object identifiers:

```
PersistenceManager pm = getPersistenceManager();
Passenger pass (Passenger)pm.getObjectById(
     Passenger.class, "" + 1);
```

By using a query, the ORMs were forced to utilize a query cache instead of a direct object identification cache, although the query cache is generally backed by the object ID cache.

The results for this test case were drastically different between the JPA and JDO implementions. Hibernate and OpenJPA were identical in processing times, while JPOX was three times slower than JPA and Speedo was seven times slower than JPA. These variances are much greater than the variances observed for the previous test case regarding the loading of a single object.

A review of the database access strategies indicated that all ORMs accessed the database only once to complete the test case for the single execution scenario, with the exception of Speedo ORM. Thus, the variations in the results are attributed mainly to the query caching ability of the ORM implementations, and secondarily to the quantity of the data returned during the partial object loading. In addition, the looping of the query compounded the differences for even the single execution of the test case.

In summary, the results of the test case indicated the JPA implementations were similar with respect to performance and query caching abilities. The JDO implementations were significantly slower, with the slower times attributed mainly to the query caching abilities of the ORM.

6.5 `LazyLoadObjects`

The results of this test case indicated the ORMs exhibited similar processing times with the exception of Hibernate which was twice as fast as the next fastest time. The average of the JPA implementations was approximately the same as the average of the JDO implementations.

The database access strategies indicate that all ORMs had the same number of remote calls, with the exception of Speedo which had nine database calls. The higher number of database calls in the Speedo ORM were related to the retrieval of set data. In Speedo, objects identified as sets were retrieved only when the object was retrieved, and not as a complete set. This is in contrast to the other ORMs which retrieved set data as complete sets. For example, in the test case, Speedo retrieved the set of `Flight` objects as individual database calls, while the other ORMs retrieved the set of `Flight` objects as a single database call. While the Speedo exhibited additional database calls, the overall processing time was

actually less for Speedo due to the object identifier lookup for the lazy loaded objects.

Similar to the other test cases, OpenJPA exhibited longer initialization times for the cache during the single execution. The cache also comes into play during the multiple execution scenarios whereby the JDO cache implementations were significantly slower when compared to the JPA ORMs.

In summary, the test case results indicated lazy object loading was relatively similar for all ORMs in terms of processing time. This was true even though the Speedo implementation executed more database calls for the single execution scenario. The performance of the caching affected the processing time of the multiple execution test cases with significant differences observed for the caching variations.

6.6 EagerLoadObjects

This test case attempted to retrieve all `PaymentStatus` data for a specific set of specific set of `Payments` with a single database call using the respective ORM interface. Thus, the test case was to return a set of `Payment` objects along with each `PaymentStatus` object associated with each of the `Payment` objects.

The Hibernate and OpenJPA implementations allowed for a simple approach to solve this test case: a JPA `NamedQuery` could be added as an annotation in the `Payment` class that retrieved the data:

```
@NamedQuery(name="getPaymentFull",
     query="SELECT p FROM Payment p JOIN FETCH
     p.paymentStatus where p.paymentAmount = ?")
```

While the above annotation is relatively easy to write and can be mapped to a name and altered without modifying the code. It also highlights the ability of the JPA implementations to generate a significant amount of SQL and hide the underlying database mapping to the ORM layer. For example, the above `NamedQuery` called `getPaymentFull` actually generates the following SQL at runtime:

```
select
payment0_.PAYMENT_ID as PAYMENT1_1_0_,
paymentsta1_.PAYMENT_STATUS_ID as PAYMENT1_5_1_,
payment0_.PAYMENT_AMOUNT as PAYMENT2_1_0_,
payment0_.PAYMENT_STATUS_ID as PAYMENT4_1_0_,
payment0_.TICKET_ID as TICKET3_1_0_,
paymentsta1_.PAYMENT_DESCRIPTION as PAYMENT2_5_1_
from AIRLINE.PAYMENT payment0_,
AIRLINE.PAYMENT_STATUS paymentsta1_
where payment0_.PAYMENT_STATUS_ID =
paymentsta1_.PAYMENT_STATUS_ID and
payment0_.PAYMENT_AMOUNT=?
```

In contrast to the JPA implementations, the JDO API did not provide a means to execute an eager load of an object graph. While JPA provides a way to detach objects from their respective `EntityManager` instances,

the detachment is performed on an object by object basis and cannot be performed in an eagerly-loaded fashion.

The processing times for this test case indicated the JPA implementations with their join statements were able to perform better than the JDO implementations. Speedo was over twice as slow as Hibernate due to the higher database calls and the initialization of the cache.

Due to the absence of a join function in JDO, these ORMs executed five database calls for the single execution while the JPA ORMs executed only one database call. As noted above, these additional calls resulted in real differences in the processing time.

Once the objects in were loaded, the caching of the objects generally leveled the performance of the ORMs during the multiple executions. The differences during the multiple executions can be attributed primarily to the initial loading of the objects into the cache. The caches, based on object caches once the IDs were loaded, tended to perform similarly in terms of processing time.

In summary, the results of this test case indicated the ability to develop complex joins through the ORM had significant affects on the processing time during the initial loading of the objects. JDO did not provide a method to execute eager loading of object graphs. The caching

generally leveled the ORMs in their respective processing times during the multiple executions.

6.7 BulkLoadObjects

This test case attempted to retrieve all `Ticket` data for a specific `Passenger` with a single database call using the respective ORM interface API. Thus, the test case was to return a `Ticket` object along with the set of `Flight` objects and the set of `Payment` objects associated with that `Ticket`. The intent was to see if the ORM can build a complex join query to bulk load related data.

The Hibernate and OpenJPA implementations allowed for a simple approach to solve this test case: a JPA `NamedQuery` could be added as an annotation in the `Ticket` class that retrieved the data:

```
@NamedQuery(
     name="getTicketFull",
     query=
          "SELECT t FROM Ticket t
          JOIN FETCH t.flights
          JOIN FETCH t.payments
          where t.passenger.passengerId = ?")
```

The above query resulted in the following complex join to be executed by both the Hibernate and OpenJPA implementations.

```
select ticket0_.TICKET_ID as TICKET1_2_0_,
flight2_.FLIGHT_ID as FLIGHT1_4_1_, payments3_.PAYMENT_ID
as PAYMENT1_1_2_, ticket0_.PASSENGER_ID as
PASSENGER6_2_0_, ticket0_.STATUS_CODE as STATUS2_2_0_,
ticket0_.TICKET_TYPE as TICKET3_2_0_,
ticket0_.TICKET_CLASS as TICKET4_2_0_,
```

```
ticket0_.BOOKING_DATE as BOOKING5_2_0_,
flight2_.FLIGHT_NUMBER as FLIGHT2_4_1_,
flight2_.AIRPLANE_ID as AIRPLANE7_4_1_,
flight2_.ORIGIN_AIRPORT as ORIGIN3_4_1_,
flight2_.DESTINATION_AIRPORT as DESTINAT4_4_1_,
flight2_.DEPART_TIME as DEPART5_4_1_,
flight2_.ARRIVAL_TIME as ARRIVAL6_4_1_,
flights1_.TICKET_ID as TICKET1_0__, flights1_.FLIGHT_ID
as FLIGHT2_0__, payments3_.PAYMENT_AMOUNT as
PAYMENT2_1_2_, payments3_.PAYMENT_STATUS_ID as
PAYMENT4_1_2_, payments3_.TICKET_ID as TICKET3_1_2_,
payments3_.TICKET_ID as TICKET3_1__,
payments3_.PAYMENT_ID as PAYMENT1_1__ from AIRLINE.TICKET
ticket0_, TICKET_FLIGHT flights1_, AIRLINE.FLIGHT
flight2_, AIRLINE.PAYMENT payments3_ where
ticket0_.TICKET_ID=flights1_.TICKET_ID and
flights1_.FLIGHT_ID=flight2_.FLIGHT_ID and
ticket0_.TICKET_ID=payments3_.TICKET_ID and
ticket0_.PASSENGER_ID=?
```

In contrast JPA, the JDO API did not provide as elegant a means to execute a bulk load of an object graph. Instead, the JDO ORMs had to execute standard queries to retrieve the objects.

The processing times for this test case indicated the JPA implementations with their join statements were able to perform better when compared to JDO. Hibernate was significantly faster, and Speedo was significantly slower when compared to the overall average times. These times, again, were generally similar when the caching mechanism was included during the multiple executions.

In summary, the results of this test case indicated the JPA implementations allowed a way to join fetch all data in the least amount of database calls. This resulted in significantly faster processing times during

the single executions, although the times were similar once the objects were loaded during the multiple executions.

6.8 BulkObjectSave

This test case examined the ability ORM in performing batch updates in a single database call, even when multiple objects had been created within a transaction. Thus, the test case was really a test of whether or not the ORM even offered the ability to execute batch inserts, and then how fast the ORM was able to execute the batch update.

The processing times for the single execution indicated Hibernate was the best performer by a four-fold margin, and the other ORMs were generally the same in terms of performance. The database access calls indicate that Hibernate was the only ORM that was able to batch insert during the execution of the test case. The overall affect of caching did not have a significant effect on the processing times since the test case was not reading data from the data base.

In summary Hibernate performed significantly faster due to the batch insert functionality offered. All ORMs performed more closely during the multiple executions as caching did not affect the overall processing times.

6.9 LoadAndSave

The purpose of this test case was to determine performance for a series of reads and a write to the database. Thus, the test case combined aspects of other test cases into a typical use case that is found in most all applications. For example, the Hibernate implementation of the test case was as follows:

```
EntityManager em = getEntityManager();
EntityTransaction tx = em.getTransaction();

tx.begin();
Passenger pass = (Passenger)em.find(
  Passenger.class, new Long(1));

Ticket t = new Ticket();
t.setBookingDate(new Date());
t.setStatusCode(3);
t.setTicketClass(8);
t.setTicketType(9999);
t.setPassenger(pass);


Query q = em.createQuery(
  "select f from Flight f where f.flightNumber =
  'XY453'");
q.setHint("org.hibernate.cacheable", true);
Flight f1 = (Flight)q.getSingleResult();

em.createQuery("select f from Flight f where
  f.flightNumber = 'AB332'");
q.setHint("org.hibernate.cacheable", true);
Set<Flight> flights = new HashSet<Flight>();
flights.add(f1);
t.setFlights(flights);

q = em.createQuery("select ps from PaymentStatus
  ps where ps.paymentDescription = 'Paid'");
q.setHint("org.hibernate.cacheable", true);
```

```
PaymentStatus ps =
   (PaymentStatus)q.getSingleResult();
Set<Payment> payments = new HashSet<Payment>();
Payment pay1 = new Payment();
pay1.setPaymentAmount(100);
pay1.setPaymentStatus(ps);
pay1.setTicket(t);
payments.add(pay1);
t.setPayments(payments);

em.persist(t);
tx.commit();
em.close();
```

The results of this test case indicated that Hibernate was the fastest performer, although OpenJPA and JPOX exhibited similar times to Hibernate. Speedo was the slowest performer, and was nearly five times slower than Hibernate.

The database calls indicated that Hibernate and OpenJPA were able to complete the test case in the least number of calls, while JPOX and Speedo required more steps to retrieve the initial data in the load phase. The processing time for Hibernate was relatively faster than OpenJPA due to the initialization of the OpenJPA cache for the required reads to implement the test case.

The multiple executions of the test case indicated that Hibernate, OpenJPA, and JPOX were relatively equal once the initial objects were read into memory. However, Speedo was significantly slower when compared to JPOX and the JPA ORMs. This difference was related to the query caching management of the Speedo implementation.

In summary, all the ORMs held similar processing times with the exception of Speedo which contained significantly slower times. The JDO ORMs required additional database calls to select the sequence, and Speedo required additional time for query cache management.

6.10 Cache vs No Cache

Caching had a dramatic affect on the ability of the Hibernate ORM to execute the loading test cases (`LazyLoadObjects`, `EagerLoadObjects`, and `BulkLoadObjects`). Similar, although not as striking results, can be seen in the other test cases for the single vs. multiple executions, as caching leveled the playing field during the multiple runs. Again, the ability of the ORM to locate the object in the local stack vs. a remote database call was very important in the processing time of the ORM in to execute the test case.

The results indicated the importance of caching in performance and the overall design decisions of the ORM. This was even seen in this study which had a locally deployed database that was running on the same node as the application. Thus, the importance of caching would become even more important given additional network overhead that most applications would occur. This additional overhead would need to be factored into the processing time of an application in the absence of a cache.

7 Conclusions and Future Work

7.1 Conclusions

This research developed an ORM performance evaluation framework by identifying a set of performance test cases that are based upon common database access scenarios. The research then executed these test cases on a selected number of Java ORMs in order to evaluate and compare ORM processing time, database access calls, and the use of object caching.

The application of the test cases revealed significant performance differences between the selected test ORMs. The JPA implementations were better performers when compared to the JDO implementations, with Hibernate being the overall best performer of all implementations. Of the JDO implementations, JPOX was the better performer, and Speedo was the overall slowest of all the implementations.

All implementations included a caching mechanism that reduced the number of database calls. Overall, the implementations were relatively similar in regards to their processing time when retrieving cached objects.

The ORMs performed equally in their respective database access strategy for loading the same object twice within the same transaction and between transactions. The differences in processing time were related to cache initialization.

Lazy object loading was relatively similar for all ORMs in terms of processing time. Eager loading objects, however, was not similar between ORMs, as JDO did not provide a method to execute eager loading of object graphs. The JPA implementations also allowed a way to join fetch related data in the least amount of database calls. Hibernate was the only ORM that offered batch insert functionality; thus, this ORM was able to perform significantly better during batch inserts.

Caching had a dramatic impact on processing time when multiple calls are made to retrieve objects. In addition, the ability to cache queries and manage the object cache with queries was important in determining overall processing time. Thus, the ability of the ORM to perform caching and possibly allow for a pluggable cache is an important decision when ORM performance is a factor in the overall application design. This research indicated that cache initialization can take a relatively significant amount of time compared to the actual database call in some situations; thus, applications should consider the initializing cache at application startup in those contexts where performance is a concern.

The results indicate that ORMs exhibit considerable differences in terms of performance and database access, while all executing the same set of test cases. These differences can be used to assist developers in navigating around performance issues in applications that utilize the specific ORM. The application of a common set of performance test cases

can be undertaken by the ORM community in order to develop a common benchmark for ORM performance and database access. Such a benchmark would provide ORM users with insight into how the ORMs implemented common data access scenarios. This information would allow developers to integrate ORM performance into decisions regarding ORM selection and the application of ORMs to specific contexts.

7.2 Future Work

The results of this research have identified additional areas of study that could be undertaken to better understand database access strategies and their associated performance implications. These additional areas of study include the following:

- Implement and execute the test cases on addition ORMs, including non-standard-based ORMs;
- Perform an evaluation of ORM caching implementations, including an analysis of distributed cache management during various update/insert scenarios;
- Add a database *insert* and *read* performance test case that evaluates ORM performance and database access for dirty objects;
- Add a database *update* performance test case that evaluates database access for updated objects in the cache;

- Add a test case for database *deletes*, including an evaluation of cascade delete operations;
- Execute the performance test cases with multiple clients connected to the same ORM cache to determine the performance variation associated with locking and synchronization; and,
- Perform the performance test cases for a wider range of JPA implementations, including the commercial versions of Oracle's TopLink and BEA's Kodo.

Bibliography

Agarwal S., Keene C., and Keller A.M. (1995). Architecting Object Applications for High Performance with Relational Databases. *Proceedings OOPSLA Workshop on Object Database Behavior, Benchmarks, and Performance*, Austin, TX, pp. 1-8.

Alur, Deepak, Crupi, John, and Malks, Dan. (2001). *Core J2EE Patterns.* Prentice Hall. 496 pages.

Ambler, Scott. (1998). *Building Object Applications That Work Your Step-by-Step Handbook for Developing Robust Systems With Object Technology.* SIGS Books/Cambridge University Press, New York.

Amber, Scott. (2005). *The Design of a Robust Persistent Layer for Relational Databases.* Ambysoft, Inc.

Anderson, T., Berre, A., Mallison, M., Porter, H., and Schneider, B. (1990). The HyperModel Benchmark. *Proceedings of the international conference on extending database technology on Advances in database technology,* Venice, Italy, pp. 317-331.

Barry, D. and Stanienda, T. (1998). Solving the Java Storage Problem. *IEEE Computer*, 31(11), pp. 33-40.

Bauer, Christian and King Gavin. (2007). *Java Persistence with Hibernate*. Manning Publications. 850 pages.

Bitton D., DeWitt D.J., and Turbyfill C. (1983). Benchmarking Database Systems, a Systematic Approach. *Proceedings of the Ninth International Conference on Very Large Data Bases*, pp. 8-19.

Carey, M. J., DeWitt, D. J., and J. F. Naughton. (1993). The OO7 Benchmark. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Washington DC., pp. 12-21.

Carey, M., DeWitt, D. Naughton, J., Asgarian, M, Brown, P., Gehrke, E. and Shah, D. (1997). The BUCKY Object-Relational Benchmark. *Proceedings of 1997 ACM SIGMOD Intl. Conference on the Management of Data*, Tucson, Arizona, pp. 135-146.

Castor User Documentation. (2007) . Castor documentation retrieved June 12, 2007 from http://www.castor.org.

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

Cattell, R. G. G and Skeen, J. (1992). Object Operations Benchmark, *ACM Transactions on Database Systems*, 17(1), pp. 1-31.

Cayenne User Documentation. (2007). Cayenne documentation retrieved 7/3/2007 from http://cayenne.apache.org/

Chays, David, Dan, Saikat, Frankl, Phyllis G., Vokolos, Filippos I., and Weber, Elaine J. (2000). A framework for testing database applications, *Proceedings of the 2000 ACM SIGSOFT international symposium on Software testing and analysis*, Portland, Oregon, pp.147-157.

Chen, Peter Pin-Shan (1976). The Entity-Relationship Model - Toward a Unified View of Data. *ACM Transactions on Database Systems*, 1(1), pp. 9-36.

Demurjian, Steven A., Hsiao, David K., Kerr, Douglas S., Tekampe, Robert C., and Watson, Robert J. (1985). Performance measurement methodologies for database systems. *Proceedings of the 1985 ACM annual conference on The range of computing : mid-80's perspective: mid-80's perspective*, Denver, Colorado, pp. 16-28.

Deng, Yuetang, Frankl, Phyllis and Wang, Jiong (2004). Testing web database applications. *ACM SIGSOFT Software Engineering Notes*, 29(5), pp. 1-10.

Doppelhammer , J., Höppler, T., Kemper, A., and Kossmann, D. (1997). Database performance in the real world: TPC-D and SAP R/3, *Proceedings of the 1997 ACM SIGMOD international conference on Management of data*, Tucson, Arizona, pp.123-134.

Duhl, Joshua and Damon, Craig. (1988). A performance comparison of object and relational databases using the Sun Benchmark. *Conference proceedings on Object-oriented programming systems, languages and applications,* San Diego, California, pp. 153-163.

Gamma, Erich, Helm, Richard, Johnson, Ralph and Vlissides, John. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley. 395 pages.

Gray, Jim. (1987). A view of database system performance measures. *Proceedings of the 1987 ACM SIGMETRICS conference on Measurement and modeling of computer systems*, Banff, Alberta, Canada, pp. 3-4.

Harkema , M., Quartel , D., Gijsen , B., and van der Mei, R. D. (2002). Performance Monitoring of Java Applications. *Proceedings of the 3rd international workshop on Software and performance*, Rome, Italy, pp. 114-127.

Hibernate User Documentation. (2007) . Hibernate documentation retrieved June 11, 2007 from http://www.hibernate.org.

Jordan, David, and Russell, Craig (2003). *Java Data Objects*. O'Reilly Press. 380 pages.

Jordan, M. (2004). *A Comparative Study of Persistence Mechanisms for the Java Platform*. September 2004. Retrieved June 20, 2007 from http://research.sun.com/techrep/2004.

JPOX User Documentation. (2007). JPOX documentation retrieved June 18, 2007 from http://www.jpox.org.

Juric, M., Rozman, I, and Nash, S. (2000). Java 2 distributed object middleware performance analysis and optimization. *ACM SIGPLAN Notices*, 35(8), pp. 31-40.

Juric, M., Rozman, I., Brumen, B., Colnaric, M., and Hericko, M. (2006). Comparison of performance of web services, WS-security, RMI, and RMI-SSL. *Journal of Systems and Software*, 79(5), pp. 689-700.

Keller, Wolfgang. (1997). Mapping Objects To Tables - A Pattern Language. *Proceedings of the 1997 European Conference on Pattern Languages of Programming*, Bavaria, Germany, pp. 1-26.

Linskey, Patrick and Prud'hommeaux, Mark. (2007). An in-depth look at the architecture of an object/relational mapper. *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*, Beijing, China, pp. 889-894.

Loney, Kevin. (2004). Oracle *Database 10g: The Complete Reference*. McGraw-Hill Osborne Media. 1369 pages.

Martin, Bruce. (2005). Uncovering Database Access Optimizations in the Middle Tier with TORPEDO. *Proceedings of the 21st IEEE International Conference on Data Engineering*, Tokyo, Japan, pp. 916- 926.

Monson-Haefel (2001). Richard. *Enterprise JavaBeans*, 3rd Edition. O'Reilly Press. 344 pages.

OpenJPA User Guide. (2007). OpenJPA documentation retrieved June 19, 2007 from http://openjpa.apache.org.

Orthogonal Persistence for the Java Platform – Specification and Rationale. Sun Microsystems Laboratories Technical Report, SMLI TR-2000-94, December 2000.

Poess, Meikel and Floyd, Chris. (2000). New TPC benchmarks for decision support and web commerce. *ACM SIGMOD Record*, 29(4), pp. 64-71.

Pole Position (2007). User documentation retrieved June 22, 2007 from http://www.polepos.org

Pugh, Eric and Gradecki, Joseph. (2004). *Professional Hibernate*. John Wiley & Sons, Inc.

Rubenstein, W. B., Kubicar, M. S., and Cattell, R.G.G. (1987). Benchmarking simple database operations. *Proceedings of the 1987 ACM SIGMOD international conference on Management of data*, San Francisco, California, pp. 387-394.

Seng, Jia-Lang, Yao, S., and Hevner, A. (2005). Requirements-driven database systems benchmark method. *Decision Support Systems*, 38(4), pp. 629-648.

Speedo User Documentation. (2007). Speedo documentation retrieved June 21, 2007 from http://speedo.objectweb.org.

Sun Microsystems (2002). *Java Specification Request 19: Java Enterprise JavaBeans 2.0 Specification*.

Sun Microsystems (2003). *Java Specification Request 153: Java Enterprise JavaBeans 2.1 Specification*.

Sun Microsystems (2004). *Java Specification Request 12: Java Data Objects (JDO) Specification*.

Sun Microsystems (2005). *Java Specification Request 243: Java Data Objects 2.0 Specification*.

Sun Microsystems (2006a). *Java Specification Request 220: Java Enterprise JavaBeans 3.0 Specification*.

Sun Microsystems (2006b). *Java Specification Request 221: JDBC 4.0 Specification*.

Tiwary A., Narasayya N. and Levy H. (1995). Evaluation of OO7 as a system and application benchmark. *OOPSLA Workshop on Object Database Behavior, Benchmarks and Performance*, Austin, Texas.

Yao, S., Hevner, A., and Young-Myers, H. (1987). Analysis of database system architectures using benchmarks, *IEEE Transactions on Software Engineering*, (13)6, pp.709-725.

Zyl, Pieter Van, Kourie, Derrick G., and Boake, A. (2006). Comparing the performance of object databases and ORM tools. *Proceedings of the 2006 annual research conference of the South African institute of computer scientists and information technologists on IT research in developing countries*, Somerset West, South Africa, pp. 1-11.