PERFORMANCE EVALUATION OF JAVA OBJECT-RELATIONAL MAPPING TOOLS

by

HASEEB YOUSAF

(Under the Direction of John A. Miller)

ABSTRACT

In the modern era of enterprise Web technology, there is strong competition growing between different vendors for highly scalable and reliable Web application frameworks. In designing these Web applications, the companies need some storage mechanism through which they manipulate the data efficiently. The traditional database management systems provide this functionality but require a complete knowledge of the SQL. Nowadays, there are many Object-Relational Mapping (ORM) persistent stores available, which make the application developer job easier by providing the same functionality as that of a traditional database. The Java language also provides ORM tools such as Hibernate, EclipseLink that provide the same functionality. Similarly, we have created a lightweight persistent tool named GlycoVault, which provides similar functionality. This study is focused on comparing different Java ORM tools including GlycoVault. The study provides information about the tools' functionalities and also compares the results of several queries we have designed to test their performance.

INDEX WORDS: Persistent, ORM, Hibernate, GlycoVault, Ebean, EclipseLink, OpenJPA, database, SQL, cache, performance.

PERFORMANCE EVALUATION OF JAVA OBJECT-RELATIONAL MAPPING TOOLS

by

HASEEB YOUSAF

BS, Iqra University, Pakistan, 2005

MBA, Iqra University, Pakistan, 2007

A Thesis Submitted to the Graduate Faculty of The University of Georgia in Partial Fulfillment of the Requirements for the Degree

MASTER OF SCIENCE

ATHENS, GEORGIA

2012

PERFORMANCE EVALUATION OF JAVA OBJECT-RELATIONAL MAPPING TOOLS

by

HASEEB YOUSAF

Major Professor:	John A. Miller
Committee:	Krzysztof J. Kochut
	William S. York

Electronic Version Approved:

Maureen Grasso
Dean of the Graduate School
The University of Georgia
May 2012

DEDICATION

To my family and friends

ACKNOWLEDGEMENTS

I would like to thank my major professor Dr. John A. Miller for helping me throughout the Masters program in increasing my knowledge and help me in completing my thesis. I would also like to thank Dr. Kochut, for his support throughout the completion of this project and providing me the suggestions and valuable ideas. I would like to say thanks to Dr. William York for giving me useful knowledge about GlycoVault and related scientific background. I would also like to thank Matthew Everson for providing me guidance in the GlycoVault source code. I would also like to thank my lab mates including Khalifeh-al-Jaddah and Ki Tae Myoung for giving me the initiative to start this research. I also want to thank my friends and family to support me during the whole masters program.

TABLE OF CONTENTS

Page

CHAPTER 1

INTRODUCTION

Most of the enterprise level applications deal with huge amounts of data. These data are presented in different formats such as XML (Xtensible Markup Language), spreadsheets and text files. This data needs to be stored or retrieved in some fashion. In object-oriented programming, these data are in the form of objects. The storage of these objects is termed as persisting the objects. Persistent can be defined as, “The characteristics of data that outlives the execution of the program that created it.” [22]

Object Persistence is defined as the storage of the objects after the execution of the program. An object-oriented language such as Java provides different persistence storage systems that provide mechanisms for persisting the data. Some of the commonly used persistence mechanisms used for persisting the data include Java serialization, Enterprise Java Beans (EJB) and Java Persistence API (JPA). These mechanisms let the data to be stored in the files or they can be stored in a database system.

However, while object-oriented languages are easy to work with regards to business modeling, the object-oriented paradigm is in contrast to database systems such as Relational Database Management System (RDBMS). The logical structure of how the data are represented in an RDBMS is quite different when compared to how object-oriented (OO) languages represent the data. Relational databases represent data in tabular form where data are logically portrayed in tables and rows. This difference in structure results in the object-relational impedance mismatch.

There are several Java persistent storage systems that provide a solution to this mismatch problem. These systems provide different techniques for mapping the domain model to the relational database. These systems also provide other features such as transaction management, query languages and caching mechanisms.

This study is focused on performance comparison between different persistent storage systems and our custom designed persistent storage system. This study provides computer scientists and developers who are mainly concerned with database driven applications to get an overall idea of what type of persistence tools are available and what features they offer.

The organization of this thesis is as follows. The second chapter discusses the related work relevant for the performance comparison. The third chapter discusses the ORM and the initial problems it faced. Chapter four discusses the persistent tools that are used for this study. Chapters five, six and seven are mainly focused on the features that these tools provide, which include query languages, transaction management and cache management, respectively. The eighth chapter presents the results of the performance comparison study. The last chapter gives the conclusions and future work.

CHAPTER 2

RELATED WORK

This chapter discusses related work that has been done for performance evaluation. These studies also discuss how different persistent architectures are tested and what benchmarks and frameworks are used to test them. These studies also show different techniques and limitations they had when comparing the tools.

2.1. ORM vs. Object DB

This study [6] was conducted to show the usage and performance of an object database in comparison with ORM tools. The tools that are chosen for this comparison are Versant ODBMS [20] and the Hibernate [1]. There are three areas that are discussed for the comparison.

The first area compares the features of the tools briefly. They provide tabular data in which different feature of each of the tools were discussed such as scalability, reliability and extendibility. The second area discusses the comparison between different features such as object-relational mapping, transaction management, caching mechanism and query languages. The third and most important area of comparison is how these tools perform. In this comparison, they use the OO7 benchmark [13] in which they perform different tests for CRUD[1] operations. For the comparison 6 different types of queries were selected. These queries are distributed as follows.

[1] CRUD stands for Create, Retrieve, Update and Delete

1 Query 1 finds 10 random single objects

2 Queries 2, 3 and 7 find a certain percentage from the AtomicPart objects.

3 Query 5 and 8 compare properties of objects between two different classes.

It is obvious from the above results that other than Q1 and Q8, Versant is the obvious winner. The reason why Hibernate is faster than Versant in the other two cases is that the query was performed on the indexed columns of objects and that is what makes Hibernate run faster. Overall, it was concluded that Versant performed better than Hibernate.

2.2. Java Persistent Store Comparison

The study in [7] conducts a comparison of different persistence mechanisms. In this study, the authors select different persistent mechanisms, such as JDBC, JOS (Java Object Serialization), JDO (Java Data Objects) and JPA (Java Persistence API), and compare the different features that were tested were orthogonally, persistence independence, reusability, performance, scalability, and transactions. Each tool is discussed separately and the above-mentioned features were discussed in detail. In order to check the performance of these tools, a benchmark was selected. The architecture of the benchmark and how it is used in the study is also discussed. In order to perform the queries, two different ranges of data were chosen. One range starts from 400 records till 4000 records. The second range start from 4000 records and ended in 40000 records. The performance was based on the traversal of the object and how the path is traversed after the id of the root object is found. Two different types of traversals were taken into consideration for test purposes. The first one is Full Traversal (basic). During this traversal, the features that were tested include the cache management system and processing time in traversing through the benchmark. The sparse traversal functionality is almost the same as full traversal system except that instead of getting the full path to be traversed, the root entity of the

path is traversed. There were two different types of runs that were taken. One is a cold run and the other is a hot run. In case of the cold runs, the objects need to be retrieved from the database every time. During the cold run, the full traversals for all the tools did not perform well except for OPJ because of persistence independence. In the case of sparse traversal, except JOS and JBP (JavaBean Persistence) who has to read the entire database, all the other tools show good performance.

In case of hot runs the tool does not has to visit the database every time to get the result, in fact in hot runs, the data are stored in memory after first run. In case of full traversal, all the persistence tools performed much better and their performance increased drastically. Most of the tests performed were based on the consideration of how the data are clustered in the persistence tool and discussed how the tools perform for incremental access.

2.3. Object Store Comparison

This study discusses different approaches to test performance of object stores [5]. This study first discusses the tools and JPA and then describes the (OO7) benchmark used to test the performance. The object stores that are being used in the comparison are Hibernate and db4o. In this study three separate criteria were chosen for performance testing.

The first phase is the creation phase. In this phase, the performance of creating new object entities and the association between them were tested. The second phase includes queries performed on the entities and how data are retrieved. There are different types of queries used to do these tests. These queries are used to find objects in the data store that matched certain criteria. These queries can be classified as follows:

- Queries on one entity type/class type, using id's to match.
- Queries with ranges.
- Queries that find all the objects of one type of class
- Queries that find objects of type A and then traverse their one-to-many associations.
- Queries that find objects of type A and do a “join” with another object B using id's to do the matching

The query results are obtained by iterating over all the objects that make up the results. The performance results are compared and discussed in regards to how well each tool performed.

It was observed that except the first and the third query, the performance of db4o was better than Hibernate. The reason being that Hibernate performs better in the case of the indexed column objects. In other cases, Hibernate takes more time in mapping tuples to objects, thus db4o performed better in those cases. Overall, it was concluded from the test results that db4o performs better than Hibernate.

2.4. JPA vs. JDO

This study compares the performance of JPA tools against JDO (Java Data Objects). Then JPA tools that were selected for this study are Hibernate and OpenJPA and for JDO, they chose JPOX [15] and Speedo [16].

There are different types of tests performed and most of the tests were based on data retrieval. The queries that were used for performance tests are as follows.

- Loading one object twice within a transaction.
- Loading one object twice between transactions.

- Lazy loading an object data.
- Eager loading an object data.
- Bulk loading similar objects.
- Bulk saving objects.
- Loading and saving complex objects

The tests were performed using the airline reservation system. One of the criteria used in the paper is processing time, which gives the direct performance results of how the ORM or persistence tool is performing for a specific test. It was observed that in the case of the Hibernate and OpenJPA, the results of the lazy loading were better in comparison to the JDO tools. Similarly in the case of second query, ORM tools performed better because of caching systems. In case of eager fetching, ORM tools have to fetch the entire object from the disk so its performance decrease in those queries. It was concluded that for most of the queries Hibernate performed the best overall, whereas, in case of queries involved eager loading and complex objects JDO performed better

2.5. Hibernate vs. iBatis

In one of the discussions, comparison between iBatis and Hibernate was undertaken. In order to perform the comparison, a small banking system was designed and different activities were included in it to perform various tests. The tests were based on a Java program, which uses both Hibernate and iBatis to perform basic SQL operations as performance test. These test were performed for single and multiple users. For multi-user tests, Java threads were used.

In order to perform the tests the following inputs can be used.

- Choose the persistent tool.

- Number of records to be inserted, updated and deleted at one time.
- Whether to perform all the operations, or perform one of the CRUD operations.
- The number of threads (users) used to perform the multiuser testing.
- The number of iterations means how many times a set has been accessed.

There are almost 5000 records that are being inserted and different conditions and filters were selected for checking performance such as increasing the number of threads or increasing / decreasing the number of iterations.

The final performance results were calculated based on the average of all averages for each operation. The results show that Hibernate takes much more time to insert data due to ORM mapping overhead and association mapping. Also, it was found that Hibernate takes more time in performing the query the first time as compared to the second time. This is mainly due to the cache implementation in Hibernate. In conclusion, it was noticed that Hibernate is slower for cold queries, while for hot queries Hibernate performs much better than iBatis.

CHAPTER 3

OBJECT-RELATIONAL MAPPING

Object-relational mapping is a technique that is used for mapping Java objects to a collection of related tuples for storage in a relational database. The class metadata for a Java object is utilized to determine the mapping. For example, consider the following Java class

```
@Entity
public class Office {
    @Id
    @GeneratedValue
    private long id;
    @OneToOne(fetch = FetchType.LAZY)
    private Employee manager;
    @OneToOne (fetch = FetchType.LAZY)
    private Employee worker;
    private String title;
    public Office () {}

    public Office (Employee m, Employee w, String t){
        manager = m;
        worker = w;
         title = t;
    } // Office
} // Office

Office office1 = new Office (…);
```

Figure 1. Java class with relevant class metadata for mapping to tuples in a relational database

Note that the Employee class is defined in the section 3.2, Figure 2. Typically, an ORM would map the `office1` object into three tuples stored in two tables (Office and Employee). See Appendix F for complete example code and representation of the data stored in the database.

Before the ORM techniques were introduced, there were some problems of mapping objects to tuples in a relational database.

3.1. ORM Paradigm Mismatch Problems

This section describes the different ORM mismatch problems that exist before useful tools such as Hibernate became available.

3.1.1. Problem of granularity

Java developers can freely define new classes with different levels of granularity. The finer-grained classes, like Address class, can be embedded into coarse-grained classes, like User class. In contrast, the type system for relational databases is limited and the granularity can be implement only at two levels: the table level (User table) and the column level (Address column).

3.1.2. Problem of subtypes

Inheritance is a widely used feature of the Java programming language that complicates object-relational mapping. Relational databases generally do not completely support type or table inheritance and if they support it, they do not follow standard syntax. For example, PostgreSQL currently supports inheritance known as table Inheritance. Inheritance is not recommended to use, because it does not support inheritance of foreign key constraint and indexes. Three mapping strategies for dealing with inheritance are discussed in section 3.3.

3.1.3. Problem of identity

The identity of a database row is expressed as the primary key. Thus each tuple in database table represents a unique record. At the object level it is not always the case. It is possible that two different objects having the same attributes values can be identified as being equal using the `equals()` function. Similarly the objects can be compared using the (==) operator.

3.1.4. Problems relating to associations

In the domain model, associations represent relationships between entities and are used for navigation. The Java language represents associations using object references, but the associations in relational databases are represented through the use of foreign keys.

3.1.5. Problem of data navigation

Accessing data and navigating from one object to another is different in Java and in relational databases. The associations between objects can be unidirectional or bidirectional as discussed. Accessing data in Java happens by navigating between objects through getter methods, e.g., `getName()`. Accessing data from the relational database and navigating from one table to another with SQL requires joins between tables and performance of the queries can be improved by minimizing the number of request to database.

3.2. Entity Mapping

An entity is a lightweight persistent domain object. An entity class maps to a table in a database and each instance of the entity corresponds to a tuple in that table. Every entity must

have an identifier, which identifies that entity. The identifier can be used on a single field or it ca composite fields. The JPA specification includes different techniques for mapping. We can use annotations or the Xtensible Markup Language (XML) mapping files to define mappings. Some of the commonly used annotations used are as follows

1. @Id. This annotation is used to define the primary key in the database table. This annotation is mandatory.
2. @EmbeddedId. This annotation is used to define the composite primary key for the table.
3. @Column. This annotation is used to define the name of the column in the database table if the name of the property/field is different from the column name. This annotation is optional.
4. @Table. This annotation is used to define the name of the table in the database to which the entity is associated with. This annotation is optional.

```
@Entity
public class Employee {
  @Id
  @GeneratedValue
  private Long id;
  private String firstName;
  private String lastName;
  @Temporal(javax.persistence.TemporalType.DATE)
  private Date birthDate;
  private String cellPhone;
  public Employee(){}
  public Employee(String fn, String ln, Date bd, String cp){
      firstName = fn;
      lastName  = ln;
      birthDate = bd;
      cellPhone = cp;
  } // Employee
```

Figure 2: ORM with annotations

3.3. Inheritance Mapping

The inheritance mapping, as the name suggests, deals with the mapping of inheritance that is defined at the object level. There are three basic types of inheritance techniques that are used to map to database schema. They are as follows.

3.3.1. Inheritance Hierarchy to Single Table – Nulls Style

In this case, a portion at the inheritance hierarchy is mapped to a single table. Typically, a class will be grouped with all of its subclasses that are part of domain model are all mapped to a single database table. Since all the entity types (classes) are mapped to a single table in the database, some of the attributes in that database table remain null. That is why sometimes this mapping approach is referred to as the nulls style [24]. In this mapping approach, there is a special type of column used known as "discriminator column" to differentiate different types of classes and subclasses used in the domain model. This is illustrated in figure 3.

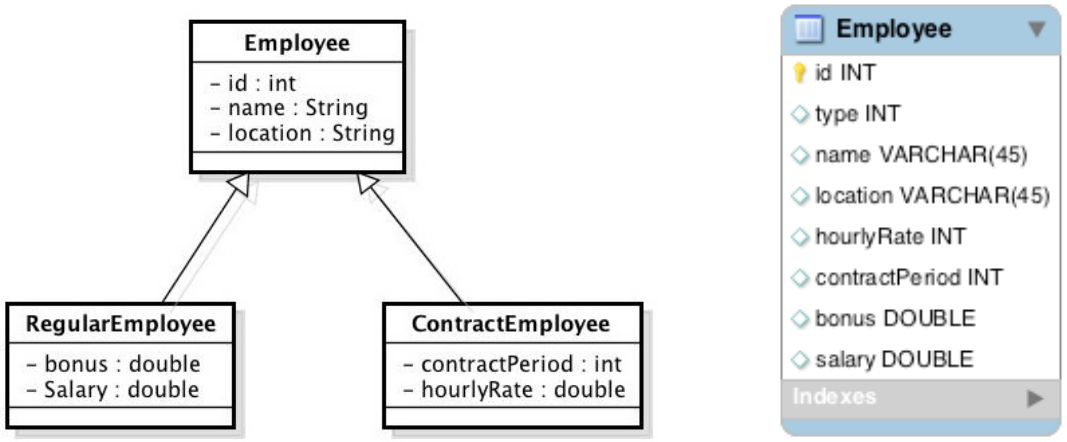


Figure 3: Null style inheritance.

In figure 3, the two subclasses and the super class on the left side of the figure are mapped to a single database table on the right side. The type column is the discriminator column that differentiates between the three classes.

3.3.2. Inheritance Hierarchy to Multiple Tables – ER Style

In comparison to the previous approach for mapping inheritance, this approach maps a portion of the inheritance hierarchy to multiple tables in the database. In this approach, all Java classes are mapped to tables in the database. Since a table is created for each entity type (class), this approach is sometimes referred to as Entity-Relationship (ER) style mapping [24]. This means that for each class/subclass created, there is a separate table created in the database. Figure 4 illustrates how this type of mapping works. For example, to persist a regular employee Java object, two tuples are created, one stored in the `Employee` table and the other stored in the `RegularEmployee` table. In order to restore the Java object, the two tables will need to be joined.

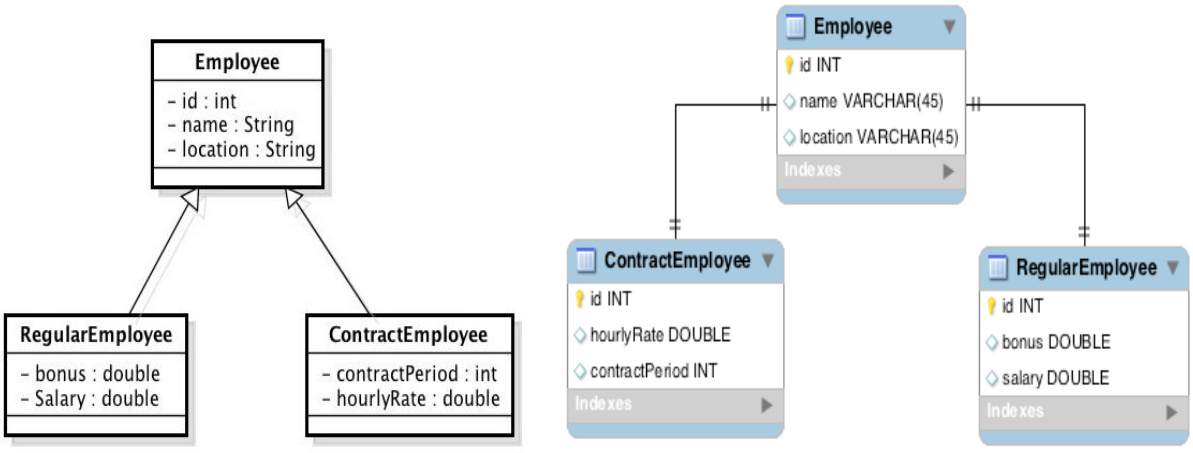


Figure 4: E/R Style Mapping

In figure 4, the three Java classes on the left side are mapped to the three database tables on the right side. The two child tables are connected to the parent table using the `Employee` primary key. This mapping approach can also specify a discriminator column. Such a column is used in GlycoVault to facilitate checking Object Constraint Language (OCL) rules for the descriptor class.

3.3.3. Inheritance Hierarchy to Multiple Tables – OO Style

In this mapping approach, each entity type (class) is mapped to a single database table. In this mapping approach, each entity type (class) maps to a separate table in the database. If the entity type is inherited from another class, it will include all the attributes from the parent class in that table. Similarly if the super class is not abstract it will be mapped to a separate table in the database. As each class is mapped to a separate table and inherits the attributes from the parent class, this type of mapping is sometimes called object-oriented (OO) style [24] mapping. Figure 5 illustrates this approach.

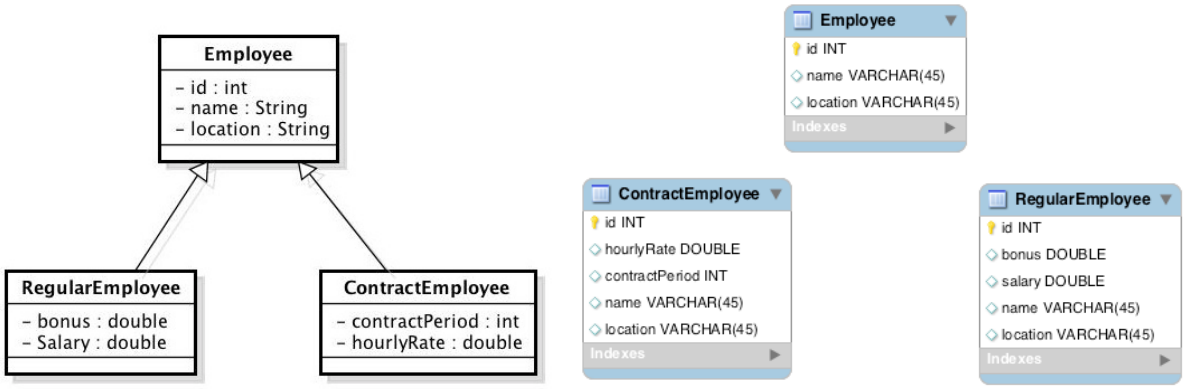


Figure 5: Object-Oriented Style Mapping

In figure 5, we can see that we have three classes on the left side of the diagram. On the right side, the tables `ContractEmployee` and `RegularEmployee` are mapped to their respective classes with inherited fields from the `Employee` class. Also, the `Employee` table maps to the `Employee` class.

3.4. Navigation Mapping

Navigation between objects in the domain model is defined based on directionality. There are two types of navigations.

3.4.1. Unidirectional Navigation

In this type of navigation, an object from only one side of association can navigate to an object on the other side, but navigation in the other direction is not supported.

3.4.1. Bidirectional Navigation

In this type of navigation, an object on either side of association can navigate to the other object. In this way the developer has more flexibility to link two objects from either side of the association.

3.5. Association Mapping

An association A on class C and D pairs objects from C with objects from D. Associations can be defined as the simple association between two classes or it can be used as the aggregation between the classes. Figure 6 provides an overview of how the two classes are associated to each other.

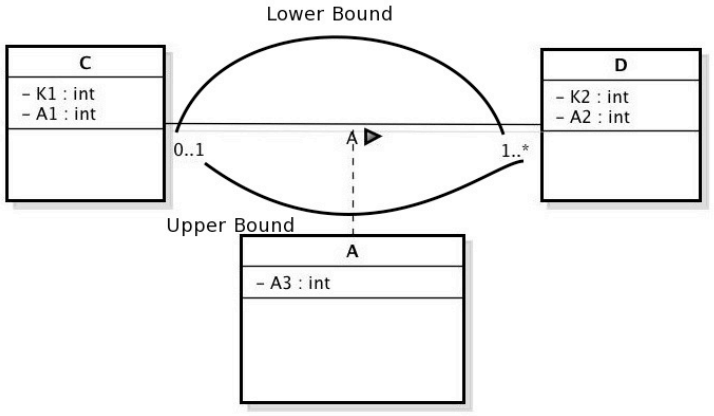


Figure 6. Association between two classes using an association class

Figure 6 illustrates the association between two classes C and D. A is the association class between these two classes. The upper arc represent the lower bound in the multiplicity and the lower arc represents the upper bound in the multiplicity. Attributes K1 and K2 are the identifiers for class C and class D, respectively, and attributes A1, A2 and A3 represent other fields.

There are three basic types of association mappings:

- **Many-to-One Association.** An association A on class C and D is many-to-one if for any object in C, there is at most one object in D, that it is associated with.
- **One-to-One Association**. If it is also the case that for any object in D, there is at most one object in C that it is associated with, A is a one-to-one association.
- **Many-to-Many Association.** If neither of the above mentioned restriction holds, it is said to be in many-to-many association.

Table 1. Association Mapping between tables and their actions

Multiplicity			Tables	Action (Update/Delete)
1:1	0..1	0..1	T1(**K1**,A1,*K2*,A3); T2(**K2**,A2)	Set Null (*K2*)
	0..1	1..1	T1(**K1**,A1); T2(**K2**,A2,*K1*,A3)	Set Null (*K1*)
	1..1	0..1	T1(**K1**,A1,*K2*,A3); T2(**K2**,A2)	Set Null (*K2*)
	1..1	1..1	T1(**K1**,A1,***K2***,A3); T2(**K2**,A2)	Cascade (*K2*)
1:M	0..1	0..*	T1(**K1**,A1); T2(**K2**,A2,*K1*,A3)	Set Null (*K1*)
	0..1	1..*	T1(**K1**,A1); T2(**K2**,A2,*K1*,A3)	Set Null (*K1*)
	1..1	0..*	T1(**K1**,A1); T2(**K2**,A2,***K1***,A3)	Cascade (*K1)*
	1..1	1..*	T1(**K1**,A1); T2(**K2**,A2,***K1***,A3)	Cascade (*K1*)
M:1	0..*	0..1	T1(**K1**,A1,*K2*,A3); T2(**K2**,A2)	Set Null (*K2*)
	0..*	1..1	T1(**K1**,A1,***K2***,A3); T2(**K2**,A2)	Cascade (*K2*)
	1..*	0..1	T1(**K1**,A1,*K2*,A3); T2(**K2**,A2)	Set Null *(K2*)
	1..*	1..1	T1(**K1**,A1,***K2***,A3); T2(**K2**,A2)	Cascade (*K2*)
M:M	0..*	0..*	T1(**K1**,A1); T2(**K2**,A2); T3(***K1***,***K2***,A3)	Cascade (*K1*,*K2*)
	0..*	1..*	T1(**K1**,A1); T2(**K2**,A2); T3(***K1***,***K2***,A3)	Cascade (*K1*,*K2*)
	1..*	0..*	T1(**K1**,A1); T2(**K2**,A2); T3(***K1***,***K2***,A3)	Cascade (*K1*,*K2*)
	1..*	1..*	T1(**K1**,A1); T2(**K2**,A2); T3(***K1***,***K2***,A3)	Cascade (*K1*,*K2*)

Table 1 above shows the combination of associations between two tables and various actions taken while performing the delete or updates on the table. T1, T2 and T3 represent the tables, the primary key are represented as bold K. If the table T1 is associated with table T2, the foreign key of the associated table will be represented in the other table in either bold italic

K if the multiplicity is 1:1 or 1:M, otherwise if the multiplicity is 0:1 or 0:M then the normal italic K is used. For the many-to-many multiplicity M:M, a third table is used with contains foreign keys (in italics) of the two tables C and D and the attribute A3 of the association table.

CHAPTER 4

PERSISTENCE STORAGE MANAGERS

This chapter discusses different Persistence Store Manager and ORM tools that are used for this study. There are five Java Persistent Managers that are used for this study. A brief description of each of these tools is given below.

4.1 JPA

The Java Persistence API (JPA) is a standard API that simplifies the programming model for entity persistence and adds capabilities for solving the ORM mismatch problem. It supports CRUD operations using the object-oriented query language called Java Persistence Query Language (JPQL). Also, it provides support for transaction management using the Java Transaction API (JTA).

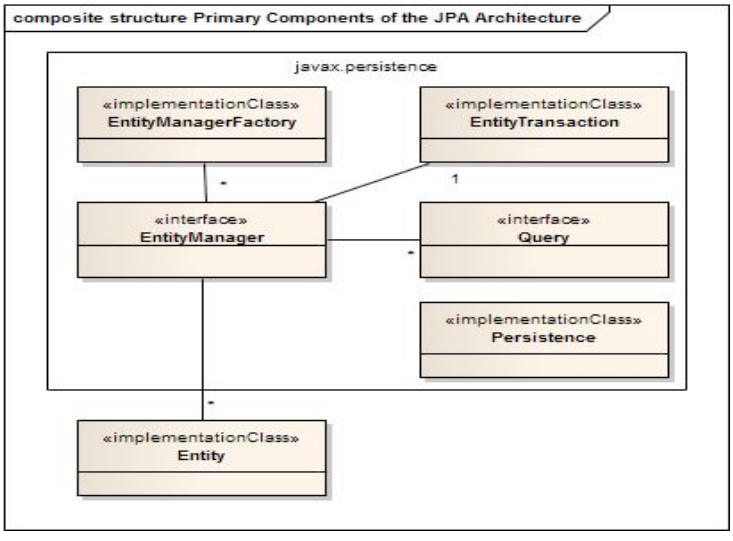


Figure 7: Class Diagram showing the relationship between different components [24]

A short description of these terms is described below.

- **Persistence.** The Persistence class provides static methods for getting the Entity Manager Factory related to the specific vendor configurations defined in the persistence.xml [20] file.
- **EntityManagerFactory.** The EntityManagerFactory class provides a factory implementation for creating an EntityManager.
- **EntityManager.** The EntityManager interface specifies methods to manage persistent objects. It also specifies methods for inserting and deleting persistent objects. The EntityManager also provides methods to prepare object-oriented queries.
- **EntityTransaction.** The EntityTransaction class has a one-to-one relationship with the EntityManager. The EntityTransaction groups all the operations in a transaction as a unit of work ensuring that all the operations either successfully commit or rollback.
- **Query.** The Query interface specifies methods for creating queries. This interface specifies methods for different types of queries such as object-oriented queries, named queries and native SQL queries.

In order to persist the entities, we need a storage area, where a set of entities can be managed It should be possible for the Entity Manager to perform operations on the entities. Such a storage area is provided by a persistence context.

A persistence context contains a set of unique instance of each entity in order to perform necessary operation on it. If multiple instances of the same entity exist in the persistence context, there can be concurrency issues.

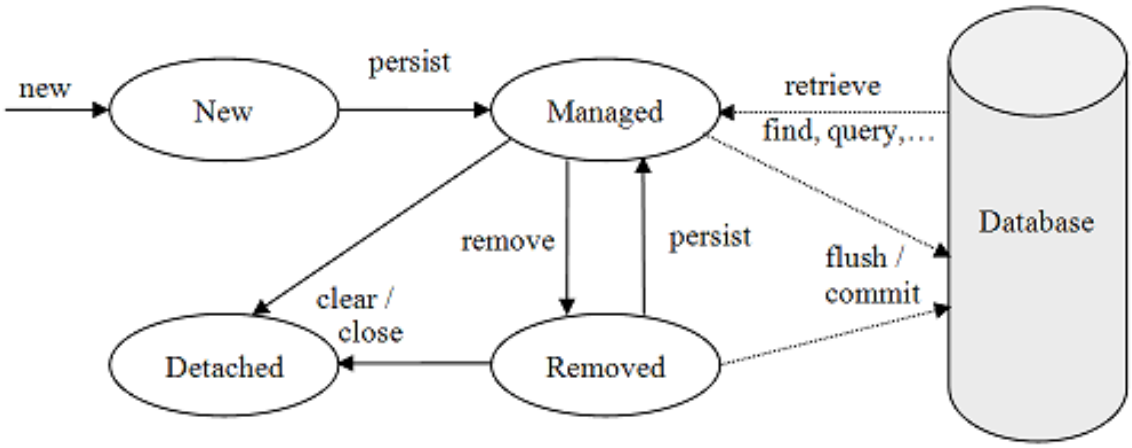


Figure 8: The Entity Life cycle in the persistence context [20].

When an entity object is initially created its state is New. In this state the object is not yet associated with an EntityManager and has no representation in the database.

An entity object becomes Managed when it is persisted to the database via an Entity Manager's persist method, which must be invoked within an active transaction. On commit, the EntityManager stores the new entity object to the database. Entity objects retrieved from the database by an EntityManager are also in the Managed state.

If a managed entity object is modified within a transaction, the update is reflected to the database on commit.

A managed entity object can also be marked for deletion; by using the Entity Manager's remove method. The entity object changes its state from Managed to Removed, and is physically deleted from the database during commit.

The last state, Detached, represents entity objects that have been disconnected from the EntityManager.

The ORM tools used in this study are as follows.

4.2 Hibernate

The JBoss Hibernate ORM framework is one of the leading ORM frameworks in the industry. It provides support for object-relational mapping as compared to traditional object database tools. The primary feature of Hibernate is mapping Java classes to relational database tables. Other features that Hibernate provides are the Hibernate Query Language (HQL), caching system and annotations for easy mapping. Similarly, Hibernate provides support of auto-generation of database schema.

4.3 OpenJPA

OpenJPA is another implementation of JPA provided by the Apache foundation. It also complies with the JPA standards. The goals of OpenJPA are to provide high performance, efficient systems and faster data access. Some of the features provided by OpenJPA are, a cache manager that uses the Apache cache system (Java Caching System)[2]. Also, it provides support for transaction handling and multiple locking mechanisms.

4.4 EclipseLink

EclipseLink [3] is one of the ORM tools that completely follow the JPA standards. EclipseLink is an open source project that was initiated by the Oracle TopLink project. In addition to providing an implementation of the JPA 2.0 specification, this ORM tool also provides support for enterprise domain models and relational schemas. One of the features that EclipseLink supports is the interface converter which lets the user choose how the database

values should be converted to the domain model and then converted back to be written in the database.

4.5 Ebean

The Avaje Ebean [4] ORM open source persistence layer is one of the simple, lightweight persistence tools that provide features of JPA, while at the same time providing the ability to write native SQL queries. One of the features of Ebean that makes it lightweight is the absence of a session manager and entity manager. Instead it has a simple Ebean and Ebean Server configuration classes. These two classes provide functionality for bypassing session and entity managers. Even though Ebean does not contain any entity manager, it has a persistence context that maintains the lifecycles of set of entity instances. Besides other features, Ebean contains a unique query language that helps developers in writing queries that closely resemble the native SQL query syntax. This API provides the developer with the ability to write SQL lite joins and conditions for getting desired results.

4.6 GlycoVault

GlycoVault is a persistent storage system that is used for storage and retrieval of experimental data in the bioinformatics area. Glycovault is a system that completely follows the object-oriented approach and provides full features for mapping Java classes into database tables. GlycoVault differentiates from the JPA specification in certain ways. Some of the differences GlycoVault has from JPA are transactions, caching systems and the fact that JPA has a query language but currently GlycoVault does not. One key feature of GlycoVault is the auto generation of the object layer and persistence layer using a JSON configuration file. This JSON

configuration file is based on the GlycoVault UML class diagram (shown in Appendix D) and contains information about entities and associations. This file can also be used in auto generation of the database schema. Figure 9 illustrates the main GlycoVault packages.

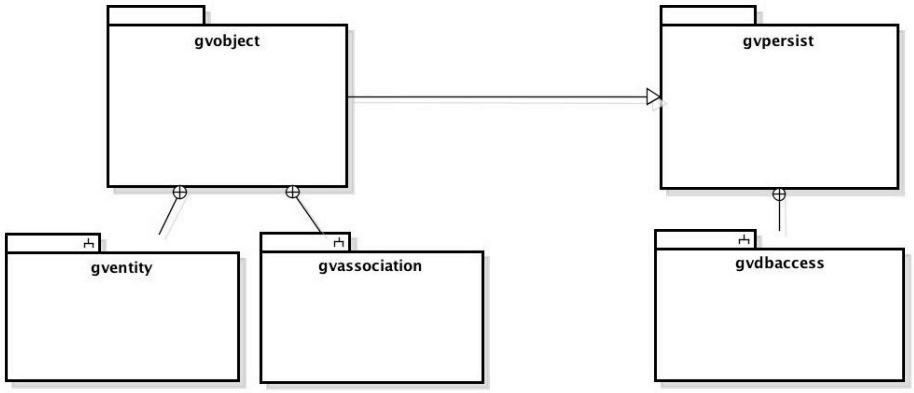


Figure 9. Package Diagram for GlycoVault

- **gvobject.** This package contains the gventity sub-package, which has the entity classes that need to be persisted. An important class in this package is GVEntityFactory, which is used to generate new entity instances. The gvassociation package is responsible for handling the associations between the classes.
- **gvpersist.** This package is responsible for creating the persistent objects. The PersistentFactory class is used to create the persistent entities. One of the sub-packages provides methods for storing, removing and restoring the entity instances. Also, this package contains another package called gvdbaccess with contains the Entity Manager and Association Manager classes with provide methods for managing entities and associations.

GlycoVault also provides other features such as a filtering mechanism, which provides support for querying the database for associations and entities. Similarly, a prototyping lightweight caching system has been implemented for faster retrieval of records.

CHAPTER 5

QUERY LANGUAGES

This chapter discusses query languages that are provided by persistence storage systems. A query language is an integral part of any database system as it provides a platform for accessing the database.

In the case of persistence systems, there are different query languages developed that help in querying and retrieval of records from the database. These query languages are explained below.

5.1 JPQL

The JPQL is query language provided by JPA for performing retrieval and other SQL operation on the entities. The JPQL syntax provides many SQL related constructs and provides support for operations such as Joins and Fetching techniques for retrieving associations and navigations in entities.

One of the differences between JPQL and SQL is that native SQL is used for operations on database systems, but JPQL operates on persistent objects and connects persistence context with the database. The second difference is independence of JPQL with the underlying database system. This means that the application developer can write the JPQL queries leaving it to the EntityManager in conjunction with the persistence context to take care of how the queries are executed at the database layer.

Figure 10 illustrates a high-level overview of how JPQL interacts between the persistence context and database system.

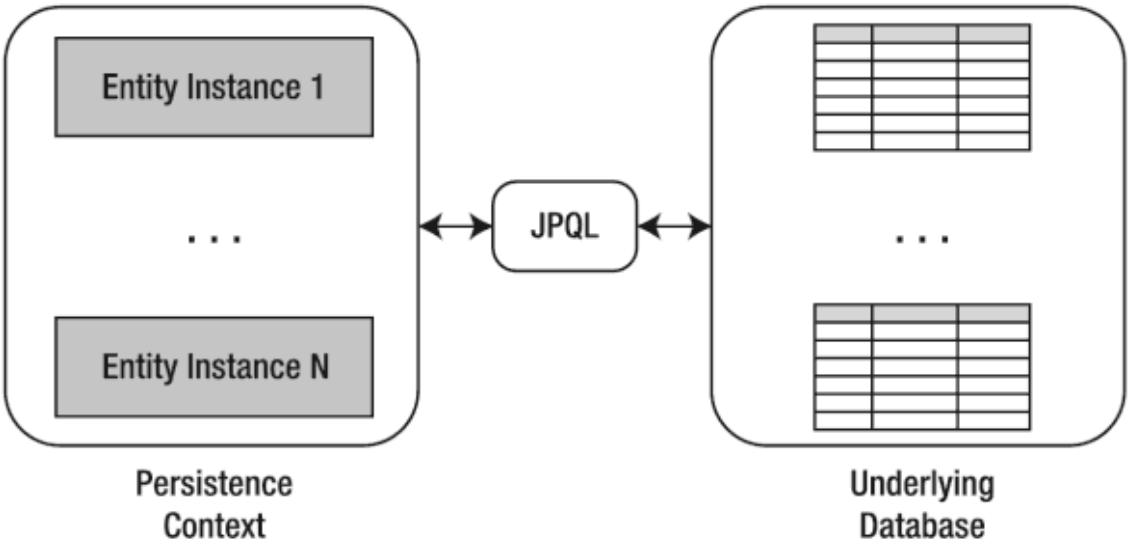


Figure 10: Abstract diagram of the JPQL Bridge [21]

JPA provides a Query Interface, which has methods for handling different types of queries. Some of the methods that are provided are listed below.

Table 2: Methods used to query data using JPQL

Methods	**Description**
createQuery(String q)	Creates a new object oriented query
createNamedQuery(String q)	Creates a static access to the object oriented query for a specific entity
createNativeQuery(String q)	Creates a query object based on the native SQL query format.
getSingleResult()	Provide a unique instance of a persistence object based on a certain condition
getResultList()	Retrieve a list of objects based on the query.
setParameter(String p, Object o)	Sets the parameters to be used in the query.

These are some of the methods provided by Query API, which help developers in managing and retrieving of data for specific entities or association between entities. For example, if we want to list the names of all the office managers.

```
q = "select o.manager.firstName from Office o where o.title = 'computer
science'")
```

5.2. HQL

The Hibernate Query Language (HQL) is an object-oriented query language, provided by Hibernate. HQL resembles SQL in many ways except that HQL works with the persistent entities in the persistence context.

Hibernate, in addition to providing an object query also provides support for named queries and SQL native queries. Named queries are useful when you want to provide static access to an entity in performing some specific task. Native SQL queries are useful in providing the developer facility to write an SQL query in a native fashion instead of an object-oriented query. The native SQL queries are very useful in cases when complicated queries need to be written and it becomes hard for the developers to write the HQL queries.

5.3. Ebean Query Language

The Ebean Query Language is a lightweight query language that is user friendly and can be easily accustomed by the developers.

One of the key features of the Ebean query language is the retrieval of Partial Objects. With this feature available, instead of retrieving the whole entity object from the database, a partial object will be retrieve that contains specific properties of the entity objects.

5.4. GlycoVault Querying

GlycoVault also provide an API for querying records in database. The query interface does not support a conventional object-oriented query language. GlycoVault provides specific interfaces, which helps in forming SQL queries. Some of the methods that are used for querying are shown in Table 3.

Table 3: Methods used in Glycovault for restoring objects

Methods	**Descriptions**
restore(Long id)	Retrieve the single instance of an object based on the id of the entity.
restoreAll(Filter filter)	Retrieve an iterators of objects based on the filter condition specified for the query. The null argument will return all the data for a specific entity.
restoreLeft(Filter filter, Object o)	This method returns an iterator for the left side entities for an association while providing information for the right side of the association and the conditions on with the entity should be retrieved.
restoreRight (Filter filter, Object o)	This method returns an iterator for the right side of the association while providing information for the other side of the association.

The GlycoVault query API also provides implicit support for other SQL operations such as JOINs, ORDER BY, Paging and range queries. One of the main interfaces that GlycoVault uses for querying is the Filter interface. This interface defines methods for providing different filters on entities.

CHAPTER 6

TRANSACTION MANAGEMENT

This chapter gives a brief description of transaction management for persistent storage systems; followed by a comparison of how the transactions are handled by the persistent storage systems used in this study.

6.1. Transaction

A transaction in database is a unit of work, which is performed on a database for a specific purpose. The main purpose of a transaction is to perform different operations in a reliable manner. The properties of transactions are given in table 4.

Table 4: Table showing different Transaction properties

Properties	Description
Atomicity	Makes sure the transaction is committed completely or is rolled back
Consistency	Makes sure that the transaction is consistent with the database rules
Isolation	Makes sure that each transaction is independent of other transactions
Durability	Makes sure that the transaction performed is durable and keeps its effect long after the end of transaction.

6.2. Transaction Demarcation

Transaction demarcation is defined as the process of opening and completing the transaction. Transaction demarcation provides programmer control of how the unit of work should be handled. It becomes the programmer's responsibility to identify the unit of work, which he wants to perform. Transaction demarcation can be performed in a bean level

transaction (alternatively given an appropriate Web development framework, transactions can be performed at the container level). The figure 11 shows an example of bean-managed transaction

```
try {
      em.begin();
      // do the desired operations …
      em.commit();
} catch (Exception e) {
      em.rollback();
      e.printstacktrace();
} // try
```

Figure 11: Transaction demarcation using bean-managed transaction

6.3. Transaction Models

The transaction models provides different types of methodologies that are used for controlling the transactions which are described as follows.

6.3.1. Local Transaction Model

The local transaction model is basically used when there is no ORM framework or Container Management System. This model is useful when the application is directly dealing with a connection and its not directly involved with transactions. The connection can be either controlled programmatically or it can be handled by the database.

GlycoVault closely follows this transaction model. In GlycoVault, transactions are performed per operation. This means that every time a new database operation is performed, the connection `autocommit()` method is set to false. Once the operation is completed, the `autocommit()` method is set to true and the operation is committed.

```
List<ScalarValueImpl> sv_list1 = Ebean.find(ScalarValueImpl.class)
                                    .fetch("task")
                                    .fetch("task.experiment")
                                    .fetch("physicalobject")
                                    .findList();
```

Figure 12. Ebean code with local transaction management

Similarly, in the case of Ebean, it also follows the same transaction model. As it does not have to explicitly define when to open the transaction and when to commit it. The above-mentioned code snippet provides an idea of how Ebean uses the static `find(Class<?> c)` method to retrieve a specific entity instance.

However, this type of transaction model is not possible in the case of frameworks or ORM tools as in these tools, the programmer must explicitly call the transaction management system. The transaction API, such as JPA EntityTransaction or JTA, internally converts the operations into SQL commands, which can then be committed.

6.3.2. Programmatic Transaction Model

In case of the programmatic transaction model, a programmer calls the transaction management system. The programmatic transaction model is used mostly in standalone ORM systems or in the applications where a developer wants to have more control over how the transactions are managed. In case of JPA compliant ORM tools, the most commonly used API is the EntityTransaction API, which gives a programmer control of how a transaction can be used. One of the disadvantages of using this model is that code is error prone as the programmer has to take care of when the transaction should begin and when to commit or rollback.

In this study, the transactions for JPA compliant ORM tools are handled using this approach. If we consider Hibernate, OpenJPA and EclipseLink, the EntityManager object will call the EntityTransaction and begin the transaction. All the operations are grouped together in that transaction as a unit of work. Once the operations are performed, the transaction is committed, which will either save that unit of work or it will be rolled back.

CHAPTER 7

CACHE MANAGEMENT

Caching is one of the most important parts of a database system. It provides temporary storage in main memory. In the case of the persistent ORM systems, the cache is used to store the persistent entities in memory, i.e., the Java objects. In most of the persistent ORM systems available in the market, there are two levels of cache defined: a first level (L1) cache and Second-level (L2) cache.

7.1. First-Level Cache

In the case of JPA compliant persistent systems, the persistence context acts as the first-level Cache. It supports the life cycle of the entity. In general, the JPA allows the first-level cache to be associated either with a transaction (the default) or a session.

7.1.1. Transactional Based Cache

In the transactional-based cache, the entities are managed until the end of the transactions. As the transaction life ends, all the entities belonging to that specific transaction are detached from the persistence context. Figure 13 gives an overview of transaction based caching.

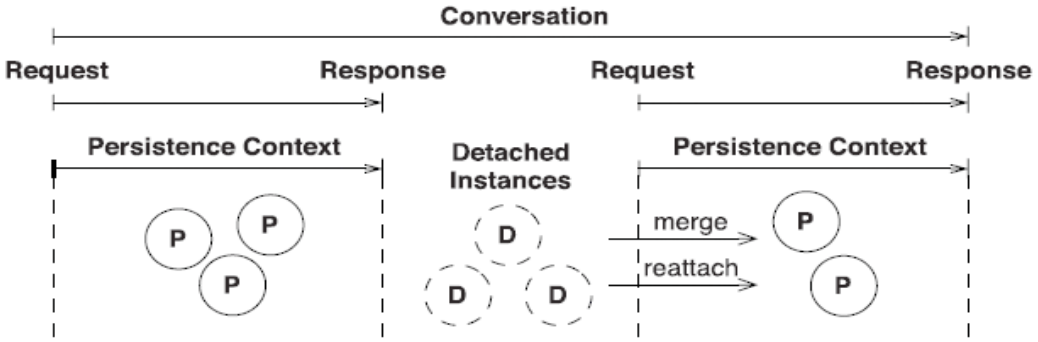


Figure 13. Transaction based caching

7.1.2. Session Based Cache

In the case of the session based cache system, the entity life cycle will remain alive throughout the session. It is also called extended session cache as the entities are not limited to a single transaction rather they are managed across multiple transactions. Figure 14 gives an overview of the session based caching.

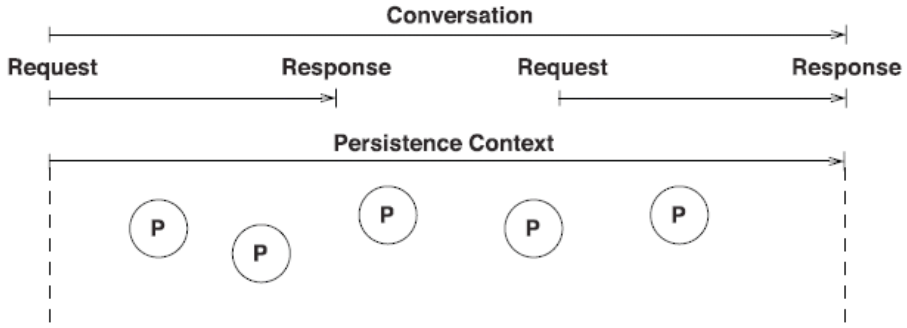


Figure 14. Session based caching

7.2. Second-Level Cache

The second-level (L2) cache is an extended cache in which the entities last through multiple persistence contexts. The basic idea behind the second-level cache is that entities do not get detached after the persistence context ends. They generally live beyond a single persistence context. The second-level cache is shared by all the transactions/sessions

7.3. Query Cache

A third type of cache that is very commonly used is the query cache. We use the query cache as it stores the results of the query in the cache along with the query string. If the user is trying to retrieve the result of the query a second time, then it will search in the cache to see it sure there is already a copy of the query. If it is available, it will retrieve the data from the cache instead of accessing the database.

7.4. Cache Comparison

In this study, we have discussed different ORM tools. These tools implement different approaches in implementing the above-mentioned caching techniques. This section discusses how the cache is implemented in different ORM tools used for this survey.

7.4.1. Hibernate

In the case of Hibernate, the first-level cache is associated with a transaction in a session, thus the entities that are executed depend upon the current session. As the transaction ends, its entities get detached from the persistence context.

In the case of the second-level cache, it is used as the performance cache. Hibernate uses an external cache called EHCache [23] for its second-level cache. The main purpose of providing the second-level cache is to share the cache across multiple transactions/sessions. The EHCache works in conjunction with Session objects in Hibernate to facilitate caching. EHCache has several properties that should be defined for each entity that needs to be cached. The code snippet in figure 15 shows the properties of EHCache.

```
<cache name=" controlledobjectImpl"
          maxElementsInMemory="10000"
          maxElementsOnDisk="1000"
          eternal="false"
          overflowToDisk="false"
          diskSpoolBufferSizeMB="20"
          timeToIdleSeconds="300"
          timeToLiveSeconds="600"
          memoryStoreEvictionPolicy="LFU"
          transactionalMode="off">
          <searchable>
             <searchAttribute name="sid"/>
          </searchable>
  </cache>
```

Figure 15. EHCache configuration file for Hibernate

If we look at a Hibernate session, it provides a method named `load(Class<Entity>, Long>` to retrieve the data from either the cache or the database. When the Hibernate session uses the load method, it will first look for the object in the session using the id provided. If it does not find the object, it will check it in the shared cache (L2 cache). If the object is not found based on its id, then it will go to the database and retrieve the data. At this point, the cache stores the objects that are retrieved from the database.

The second time when the load method is used to get the object, it will check based upon the object's id whether the object is in the cache; if the id is found, then it does not have to access the database; rather it loads the object from the cache.

JPA 2.0 provides an additional cache API, which is dedicated for the second-level cache. It provides methods for eviction, and to check whether an object is available in the cache.

In the case of the query cache, Hibernate used two components to store the query information. The first part of the query cache uses the StandardQueryCache API to store the actually query string along with ids of the entities that are used to get the rest of the data. The second part is the UpdateTimeStampCache, which will keep the query cache updated and evict the data from the cache if the data in the cache is stale.

7.4.2. EclipseLink

In the case of EclipseLink, the first-level cache also has the same mechanism as the Hibernate cache but different terminology. EclipseLink has the object cache, which can keep the entity persistent though out the transaction and in the case of the shared cache, the entity does not depend on the transaction to stay alive, instead the entity remains active throughout the EntityManagerFactory or the Persistent Unit.

In the case of the L2 cache, the EclipseLink has a slightly different approach than Hibernate. In EclipseLink, there is no dehydrated state maintained by the cache; instead, the whole entity is stored in the cache. The life of the entity in the cache depends on what is the cache type set for that entity. The Cache API can be accessed using XML mapping file or by using annotation in the entity/association class. The cache type attribute defines how the entities should be maintained in the cache.

Some of the properties are given in table 5.

Table 5: Attributes for the cache in the EclipseLink

Attribute	Description
Full	The entity object will remain in the cache until it is manually evicted or removed
Weak	The entity object will be garbage collected if it is not referenced for a specific time
Soft	The entity object will hold a soft reference and will be garbage collected if memory is low
No Identity	The entity object will not hold any reference in the memory and cannot be used in cache.

In the case of the query cache, EclipseLink uses the ReadQuery API to read the data once the query is executed on the database. As in this case, the query will index the cache to search for the query string with the same parameters. If the query found a match in the cache, then it does not have to go to the data source to retrieve the results and can get it directly from the cache. The ReadQuery API can be configured to cache the resultset of a query. It can also be configured to store a specific number of result sets in the cache.

7.4.3. OpenJPA

In the case of the Apache OpenJPA, the concept for the first-level cache is almost the same as in the case of EclipseLink. However, in the case of the L2 cache, also known as the data cache, the entities are stored in the cache when they are committed or when they are loaded from the data source. The data cache will keep the full reference for the objects and if they are updated, they have to be manually invalidated or deleted from the cache.

In the case of the query cache, the query will look for a specific id or set of ids based on the query parameters used in the query and the query string. If there is a match found in the cache, then the data are loaded for that query. The query cache is mostly used in the cases where the data are not frequently changed.

7.4.4. Ebean

In the case of Ebean, there are two types of caches used. One of them is the bean cache which works in a similar way as that of the EntityManager / Session. Thus, in order to use the cache to retrieve the entity, the user has to set some of the cache attributes on the entity. One important attribute is readOnly, which is used to retrieve the less frequently updated entities from the cache.

The second type of cache used is the query cache. Ebean has been specifically designed this cache to be used with infrequently updated data. The query cache is applied mostly on lists and sets.

7.4.5. GlycoVault

GlycoVault provides supports some caching. In GlycoVault, there is only one level of cache implemented. As discussed before, the cache implemented in GlycoVault is basically at the session level. As soon as the user is logged in to the system, a new session is created for that user which also contains the information related to the connection and access group with which it belongs. When this session is created, the user can optionally enable or disable the cache.

The cache is basically used in the EntityManagerImpl and the AssociationManagerImpl class. We have defined a class called GVCacheManager that manages different cache operations. GlycoVault makes use of the third party cache tool called the EHCache, which is widely used in most of the commercial persistence tools including Hibernate and OpenJPA.

The elements in the cache are in the form of key-value pairs, which contain the ids of the entities and their respective values. When the query is executed the first time, the entity ids related to the entities are first retrieved from the database. These entity ids are then checked

against the cache. If the cache does not contain the id, the entity is saved in the cache. If the same query is run again to retrieve the same resultset, the same procedure is applied and as the cache contains the result, the entities are retrieved based on the ids in the resultset. Similarly, if new entities are added to the database the relevant part of the cache will not be invalidated, but rather they will be retrieved from the database. If an id is not in the cache, the entity related to that id will be retrieved from the database and inserted into the cache. It should be noted that the caching implemented in GlycoVault is currently just an exploratory prototype.

CHAPTER 8

PERFORMANCE COMPARISON

This chapter compares the different ORM persistent stores used in this study. In order to perform the comparison tests, we have to design a set of queries, which are going to test different features of the ORM systems which are discussed in this study.

8.1. Domain Model

In order to check the performance of GlycoVault and other persistence stores we need to understand its domain object, which specifies how the different components are attached and how the flow of data is performed. Appendix E has the UML class diagram for GlycoVault.

The UML class diagram presents three different views: 1) Experiment Design View 2) Experiment Setup View and 3) Experiment Execution View. The Experiment Design View presents the design point of view of GlycoVault. An experiment usually starts with some initial sample, which is represented by the “SourceSample” class. A “DerivedSample” class is used to represent those samples, which are produced at the end of an experiment step or task. We also extract some important information represented by “Observable” instances, from data sources and store them in GlycoVault database. The extracted data are stored in the “ScalarValue” class. These values can be either strings or floats. The “Vector” class is used to represent the order in which these scalar values are stored. Scientists usually carry out experiments, so we have a

"User" class to represent them. The "ControlledObject" class is the parent class of all the classes. ControlledObject is also used to provide access rights to the user.

8.2. Evaluation Setup

In order to study the performance of query processing, we have to setup an experiment. For setting up the experiment, we have used the transcript processing data (MolecularObject), which is available to us through a spreadsheet file. For our evaluation setup, we have approximately 2241 transcript profiling records available. Based on this data, we selected the number of experiments to produce the experimental data. For example, in order to generate approximately 80,000 records of data, we have used 20 experiment objects. Each experiment will use all the molecularObjects and thus each molecularObject will in turn be associated to a specific scalar value, which have observables (in our case Average and Standard Deviation). Thus, each experiment will contain 4482 Scalar Values in this case.

In order to conduct the evaluation, we have chosen a machine with the following configurations:

- Operating System. Ubuntu Linux 11.10 (64 bit)
- Process: Intel Core 2 Duo CPU E7400@2.8GHz x 2
- RAM: 4 GB
- H.D.D: 200GB

8.3. Test Queries

One of the important factors when doing a comparative study is to select a benchmark or set of test queries. This purpose of the benchmark is to check different features of a tool and to compare the overall results of the systems. There are several benchmarks available for

performing different types of tests and to check the performance of different tools. One of the benchmark that is very commonly used is the OO7 benchmark. GlycoVault is primarily designed for the purpose of retrieving experimental data related to bioinformatics. The OO7 benchmark is related to engineering and thus is not ideal for our evaluation.

In order to perform the Test we have decided to use five different queries, which check the performance of systems from different point of views. These queries are listed below:

1. Given an Experiment ID, get the experimental data.
2. Given the ExperimentSetup, get the experimental data.
3. Given the protocol, get the experimental data.
4. Given the source sample, get the experimental data.
5. Given the source sample and experimental design (experiment setup), get the experimental data.

The SQL and HQL version of these queries are shown in Appendix E. Also, the path used to JOIN the objects for each query is also shown in Appendix D.

8.4. Evaluation Criteria

We evaluated the results based on the following criteria.

- The processing of each query is conducted exactly the same way for all the persistent storage systems.
- The queries are run 10 times one after the other.
- We start the evaluation with 5000 records and then increase the records exponentially (10000, 20000, 40000, 80000, 160000).
- For each data set size, a separate schema instance is created.

- For testing each set of data, we randomly choose the parameters that are used to perform the query.
- The average and the standard deviation are calculated for each type of query and for different ranges of data.

8.5. Comparison Results

The comparison results are presented in the form of tables and graphs. The tables are distributed on the basis of the number of records retrieved. All the times provided are in milliseconds.

Table 6: Table containing the averages of queries on different tools for 10000 records (time in ms)

Tools / Queries	Glycovault	Hibernate	EclipseLink	OpenJPA	Ebean
Query 1	1023.23	1673.49	4367.39	3700.23	1375.23
Query 2	1202.23	1703.84	4194.99	4023.23	1833.86
Query 3	1132.23	1656.10	4158.85	3934.23	1544.59
Query 4	1632.34	1642.24	4308.85	4290.23	1834.34
Query 5	1939.23	1964.68	5251.58	4523.23	1994.89

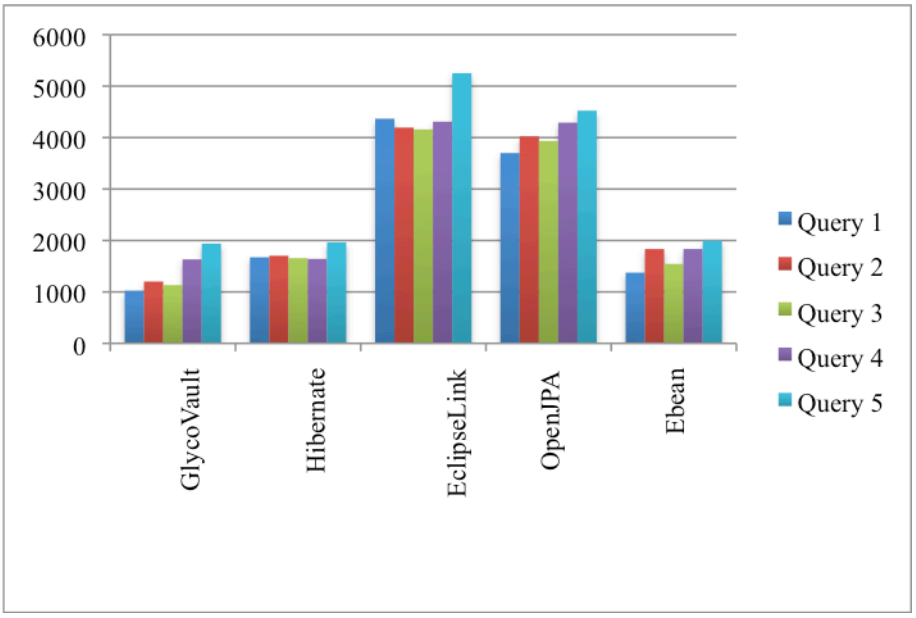


Figure 16: Graph showing averages for 10000 records for persistent tools (time in ms)

Table 7: Table containing the averages of queries on different tools for 80000 records (time in ms)

Tools / Queries	Glycovault	Hibernate	EclipseLink	OpenJPA	Ebean
Query 1	15161.07	20802.24	25233.23	41034.23	16327.47
Query 2	18172.87	23070.23	29343.23	44567.54	18203.32
Query 3	17381.00	21331.21	27990.23	43436.42	17270.31
Query 4	22942.24	25731.25	38932.23	49432.45	20423.35
Query 5	26313.12	28132.23	43999.23	56467.42	24114.23

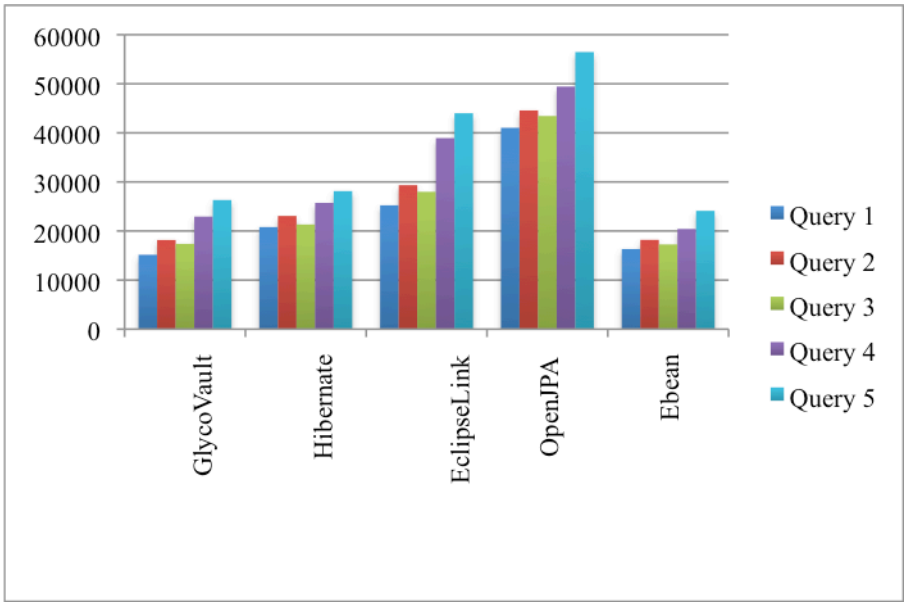


Figure 17: Graph showing averages for 80000 records for persistent tools (time in ms)

Table 8: Table containing the averages of queries on different tools for 160000 records (time in ms)

Tools / Queries	Glycovault	Hibernate	EclipseLink	OpenJPA	Ebean
Query 1	24170.82	30980.41	33233.23	69423.21	24715.23
Query 2	28290.27	34402.52	38234.63	71340.35	24842.25
Query 3	27124.23	31254.25	35642.43	70932.32	24492.32
Query 4	32123.14	36347.98	44346.36	78232.23	32650.24
Query 5	34987.78	40224.25	51998.34	83323.23	34772.25

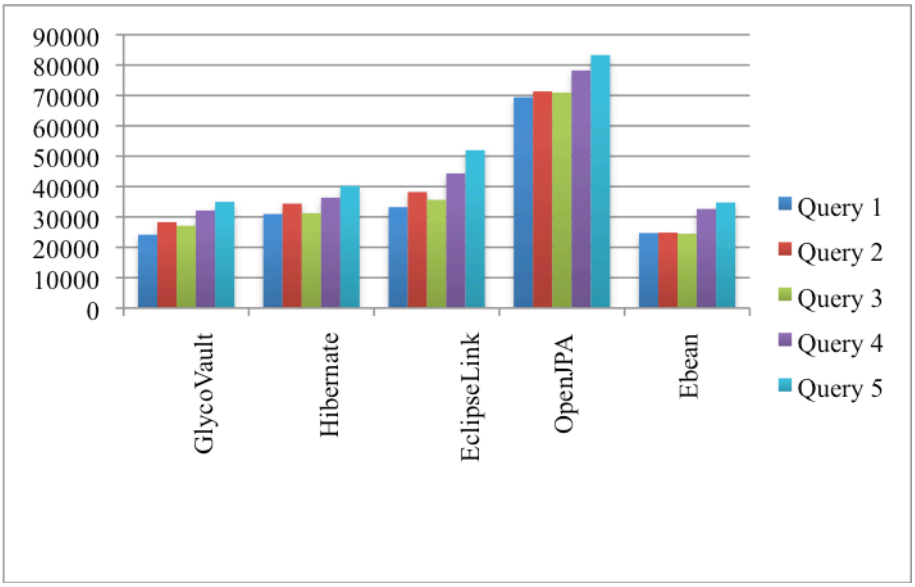


Figure 18: Graph showing averages for 160000 records for persistent tools (time in ms)

Table 9: Table containing the averages of queries on different tools for 5000 records with cache (time in ms)

Tools / Queries	Glycovault	Hibernate	EclipseLink	OpenJPA	Ebean
Query 1	376.06	533.34	826.24	826.12	568.65
Query 2	556.67	683.24	944.53	949.45	597.12
Query 3	376.96	553.92	889.98	840.23	559.58
Query 4	696.40	769.25	862.34	1038.23	935.63
Query 5	821.34	899.57	844.12	1219.23	1053.72

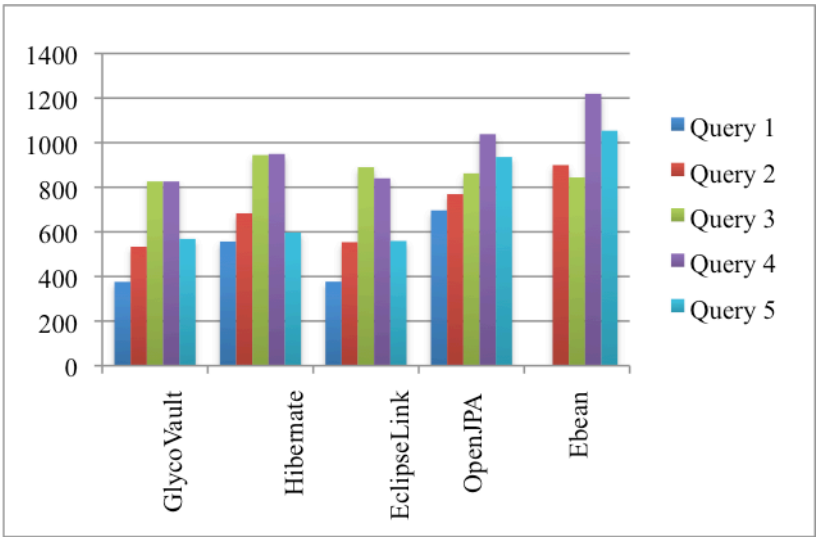


Figure 19: Graph showing averages for 5000 records for persistent tools with cache (time in ms)

Table 10: Table containing the averages of queries on different tools for 80000 records with cache (time in ms)

Tools / Queries	Glycovault	Hibernate	EclipseLink	OpenJPA	Ebean
Query 1	4355.69	1806.23	2992.12	5686.32	2986.97
Query 2	4685.95	2187.79	3490.23	5972.44	3643.23
Query 3	4204.29	1977.98	3270.43	5480.46	3272.34
Query 4	5054.56	2765.98	3990.23	6440.23	4323.23
Query 5	5632.23	2990.45	4430.26	7987.21	4804.43

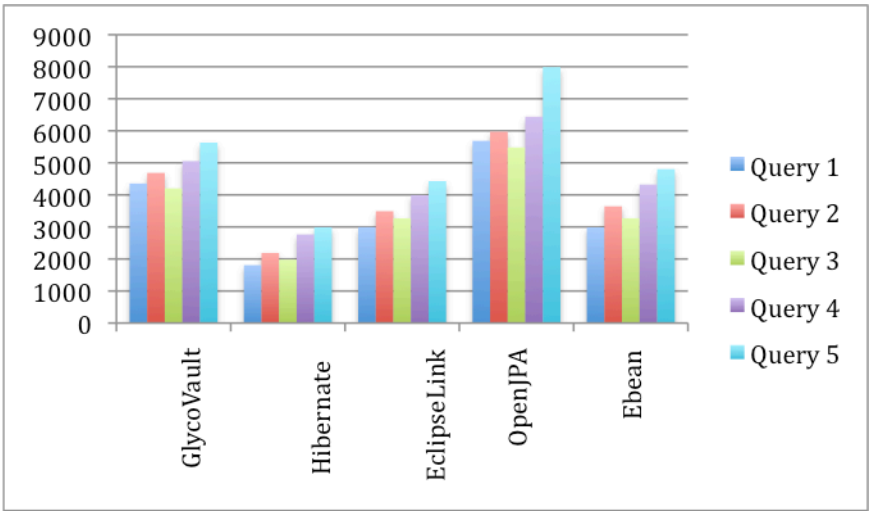


Figure 20: Graph showing averages for 80000 records for persistent tools with cache (time in ms)

Table 11: Table containing the averages of queries on different tools for 160000 records with cache (time in ms)

Tools / Queries	Glycovault	Hibernate	EclipseLink	OpenJPA	Ebean
Query 1	7696.05	2890.97	5523.32	7324.23	5686.98
Query 2	9671.70	3275.34	5743.12	7953.34	4997.96
Query 3	8423.98	2901.98	5210.32	7267.23	6198.87
Query 4	10557.91	3889.89	6370.54	8222.32	6432.23
Query 5	11434.25	4196.87	6933.12	8465.45	5686.98

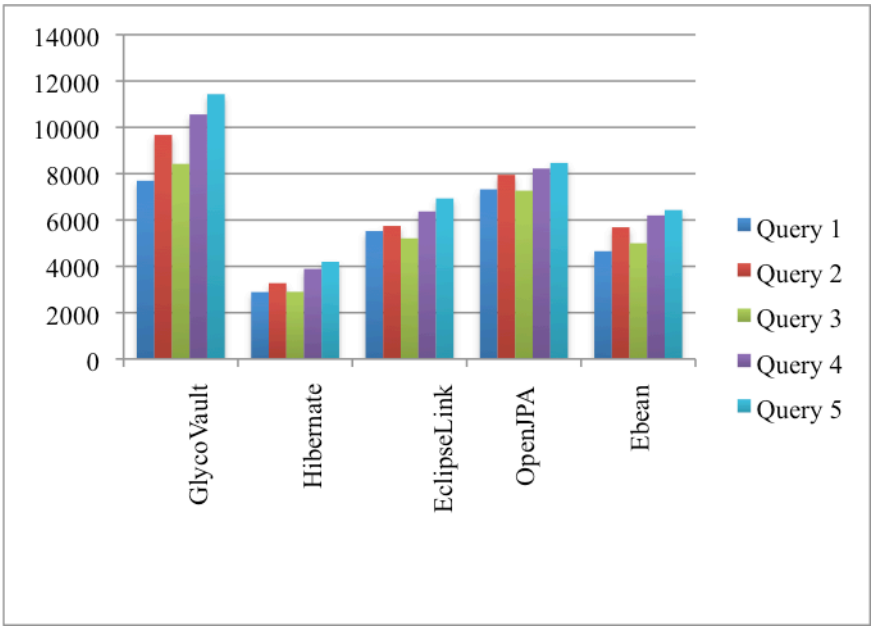


Figure 21: Graph showing averages for 160000 records for persistent tools with cache (time in ms)

If you compare the results of uncached with the cache results, it is clear that in the uncached version of testing, GlycoVault is the obvious winner in the larger dataset. As in the case of smaller datasets such as 5000 and 10000 records (shown in Appendix E), the results are almost the same in all the tools, but as the dataset gets larger (40000, 80000 and 160000) the performance of the other tools in comparison to the GlycoVault is less, except the case of Ebean in which case it showed very competitive results. One of the reasons for that can be that Ebean does not completely follow the JPA standards and provides some of its own methods for querying and managing the data (explained in Chapter 5). Also, Ebean does not provide all the inheritance mapping specially E/R style (see chapter 3 section 3.3.2). It only supports Nulls style inheritance mapping (see chapter 3 section 3.3.1). Among the JPA persistent tools, Hibernate is the winner in all the tests with OpenJPA being the worst competitor. Now, if we look at the results for the cached version, in the case of 5000 dataset, GlycoVault is again the winner, but with the minor differences from other tools. However as we increase the sizes of dataset, the time

taken to perform the queries will be more than in comparison to Hibernate. The reason being that cache mechanism (see chapter 7 section 7.4.5) that is used in GlycoVault is less efficient than the one used by Hibernate. In the graph illustrated in figure 21, we can see that time it took for Hibernate to perform and compared to GlycoVault, was less than half the time. It is obvious that Hibernate caching performs better when it comes to large datasets as compared to GlycoVault. In the case of Ebean, again the results are better than expected because the lightweight nature of the tool. The complete results for all the datasets are provided in Appendix D.

Also, to further compare the performance of GlycoVault, we also ran the test queries using the JDBC API and PGAdmin. We have chosen only 20000 and 40000 records dataset for this evaluation. The reason being, we wanted to get an idea of how the performance differs from the ORM tools. The results of JDBC API are given below.

Table 12. Comparison of results between GlycoVault, JDBC and PGAdmin for 20000 records (time in ms)

Test / Queries	**PGAdmin**	**MEAN (JDBC)**	**MEAN (GlycoVault)**
Query 1	628	1792.19	1822.22
Query 2	665	1677.22	2194.12
Query 3	236	1532.34	1935.43
Query 4	654	1579.32	2323.11
Query 5	1561	2120.34	2745.65

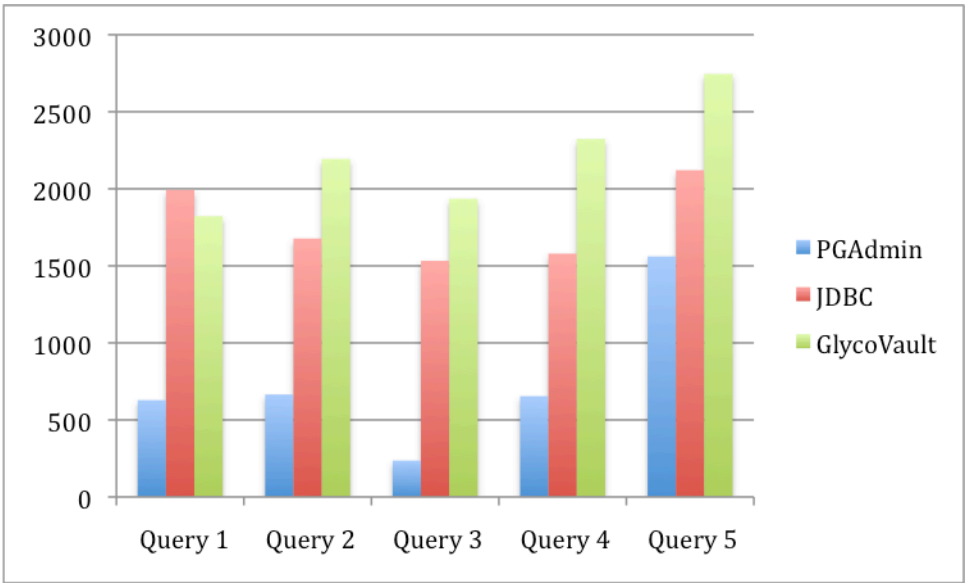


Figure 22. Graph showing Comparison of results between GlycoVault, JDBC and PGAdmin for 20000 records (time in ms)

Table 13. Standard Deviation for 20000 records using JDBC API

Test / Queries	**Standard Deviation**
Query 1	558.44
Query 2	151.67
Query 3	222.23
Query 4	177.6
Query 5	319.69

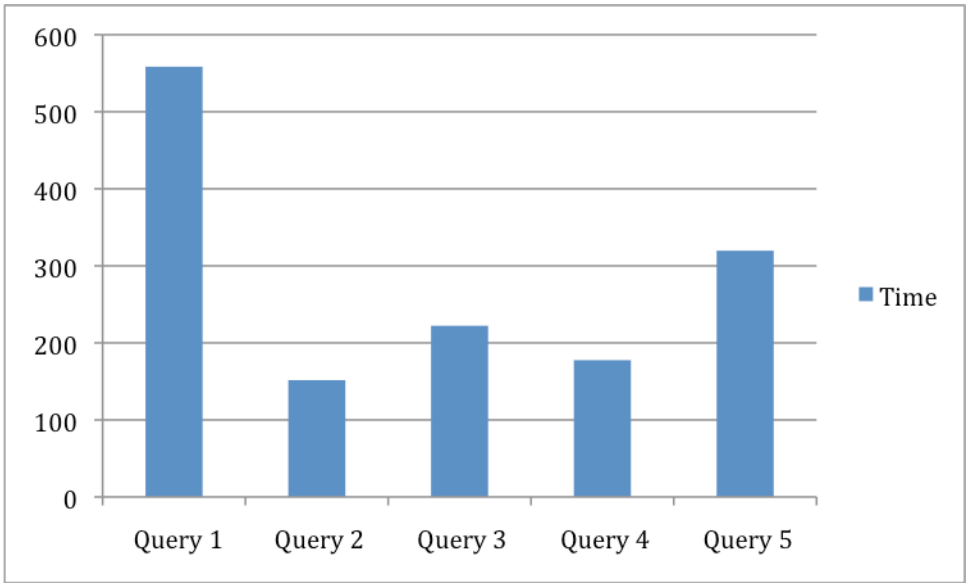


Figure 23. Graph representing Standard Deviation for 20000 records using JDBC API

Table 14. Comparison of results between GlycoVault, JDBC and PGAdmin for 40000 records (time in ms)

Test / Queries	PGAdmin	MEAN (JDBC	MEAN (GlycoVault
Query 1	1043	5660.34	6883.97
Query 2	998	4712.56	10601.29
Query 3	979	4224.78	9437.22
Query 4	1032	4414.82	10583.15
Query 5	1121	5842.67	13031.32

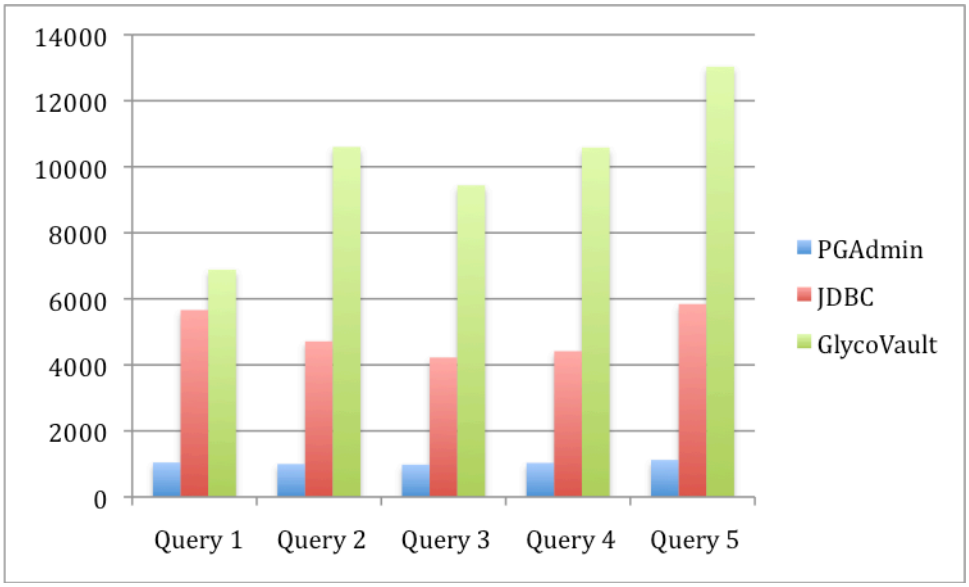


Figure 24. Graph showing Comparison of results between GlycoVault, JDBC and PGAdmin for 40000 records (time in ms)

Table 15. Standard Deviation for 40000 records using JDBC API (time in ms)

Test / Queries	**Standard Deviation**
Query 1	699.32
Query 2	523.23
Query 3	534.35
Query 4	775.35
Query 5	679.34

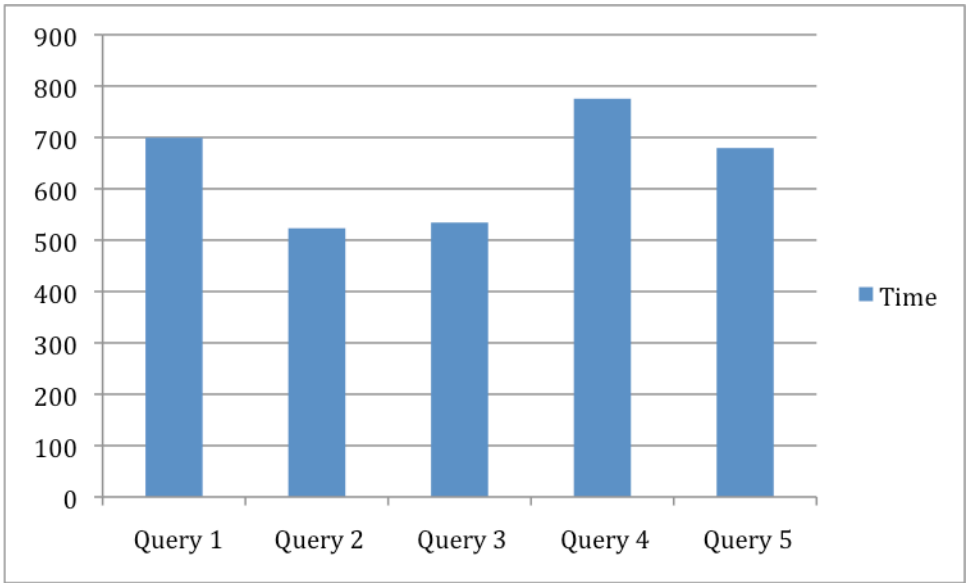


Figure 25. Graph showing the standard deviation for 40000 records using JDBC API (time in ms)

If you consider the figures related to the results of JDBC API and PGAdmin, we can see that time to execute the queries in PGAdmin are much lesser than any of the persistant stores results that are compared without caching.

Thus, based on the overall results we have observed that GlycoVault performed better overall in the case of uncached queries, while Hibernate and Ebean being close competitors. In the case of cached comparison results, Hibernate outperformed all the other tools.

CHAPTER 9

CONCLUSIONS AND FUTURE WORK

This chapter concludes the survey for performance comparison of ORMs. It also discusses future work related to GlycoVault and how we can add new features to improve its performance.

9.1. Conclusions

This section gives the conclusions for this performance study. All of the persistence storage systems that we have tested provide different features with own their pros and cons. As this study basically focuses on how each system performs while retrieving the data, it is important to focus on the factors such as cache management, query language and the ORM features that these tools have. Similarly, there are different transaction management techniques implemented by the systems that can affect the performance of the system.

For our survey we used different types of queries to check how the different systems perform. These query are selected based on different criteria and navigation paths. These queries provide a way to check the different features of the systems such as their query languages, the inheritance strategies, and their cache mechanisms. These features can help us decide which system performs better and what needs to be done to increase performance. Thus, we can conclude that no system is designed perfectly and all the system needs improvement on one feature or the other.

9.2. Future Work

This section focuses on the future work that needs to be done on the system that we have developed to make it more efficient. Some of the potential areas that we are planning to work on are as follows.

9.2.1. Query Language

Currently, GlycoVault provides support for queries in the form of user inputs and filters. It does not support as complete query language as the other tools do. In our comparative study, we observed that all the tools that we compared with our system support a richer query language with much of the functionality of the SQL language. Thus, we are planning to provide a query language in the future, possibly one similar to Ebean.

9.2.2. Second-Level and Query Cache

Currently, the cache system that we implemented in GlycoVault does not support a second-level cache or a query cache. It is just a exploratory prototype. The query cache is useful in caching the result set of the queries. Similarly a second-level cache goes beyond the scope of the session and can last even if the session is expired. Thus, we are working on caching to make the system more efficient.

9.2.3. Future Performance Evaluation

We have also planned to do some future comparative studies for our tool based on new features, including a new cache system. Also, the current comparative study is done using

PostgreSQL database management system. It might be interesting to see how these tools perform using other open source database management system such as MySQL, SQLlite and MonetDB.

REFERENCES

1. Hibernate Java Documentation; http://docs.jboss.org/hibernate/orm/4.1/javadocs/
2. Apache OpenJPA User Guide 2.3; http://openjpa.apache.org/builds/latest/docs/docbook/manual/main.html; accessed Nov. 2011
3 EclipseLink Documentation Center; http://wiki.eclipse.org/EclipseLink/Documentation_Center; accessed Oct. 2011
4. Ebean Documentation; http://www.avaje.org/ebean/documentation.html; accessed Aug. 2011.
5. Zyl Pv, Kourie DG, Boake A. 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: South African Institute for Computer Scientists and Information Technologists; p. 1-11. 2006
6. Kopteff M. The Usage and Performance of Object Databases compared with ORM tools in a Java environment; Proceedings of the 2008 ICOODB International Conference on Objects and Databases; Berlin, Germany; 2008.
7. Mick Jordan. *A Comparative Study of Persistence Mechanisms for the Java™ Platform*; Technical Report. Sun Microsystems; Inc, Mountain View, CA, USA; 2004
8. Narasayya N. and Levy H. Evaluation of OO7 as a system and application benchmark. OOPSLA Workshop on Object Database Behavior, Benchmarks and Performance, Austin,

TX, October 1995.

9. Sun Microsystems, Inc. Java Object Serialization Specification; http://java.sun.com/j2se/1.3/docs/guide/serialization/spec/serialtoc.doc.html, accessed Jan 2012
10. Sun Microsystems, Inc. JSR-000012 Java TM Data Objects (JDO) Specification; http://jcp.org/aboutjava/communityprocess/final/jsr012/index2.html, accessed Jan. 2012
11. Trinh TT. Migration from legacy Persistence API to JPA (Java Persistence API) based Implementation [Thesis]: University of Applied Sciences Hamburg; 2008.
12. Kevin R. Higgins. An Evaluation of the Performance and Database Access Strategies of Java Object-Relational Mapping Frameworks [Thesis]: University of Kansas; October 5, 2OO7
13. CAREY, M. J., DEWITT, D.J, NAUGHTON, J. F. The OO7 Benchmark; In Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data, Washington, D.C, United States, May 1993, PETER BUNEMAN AND SUSHIL JAJODIA, Eds. ACM Press, New York, NY, USA, 12-21.1993
14. Versant Documentation; http://www.versant.com/products/versant_database_engine.aspx. accessed, Dec. 2011
15. JPOX Documentation; http://sourceforge.net/projects/jpox/?_test=beta, accessed Oct, 2011
16. Speedo User Manual; http://speedo.ow2.org/doc/usermanual.html, accessed Oct 2011
17. Santiago Rodríguez. JPA implementations comparison: Hibernate, Toplink Essentials, Openjpa and Eclipselink; http://terrazadearavaca.blogspot.com/2008/12/jpa-implementations comparison.html; accessed Jan 2011.
18. Georges Polizois; Hibernate Caches; http://acupof.blogspot.com/2008/01/background-hibernate-comes-with-three.html; accessed Feb 2012

19. Hibernate JIRA “Long “in” lists in query when parsing the resultset https://hibernate.onjira.com/browse/hhh-2166**;** accessed Jan 2012

20. Entity Life Cycle; http://www.objectdb.com/java/jpa/persistence/managed; accessed Feb, 2012

21. Vasiliev Y. *Beginning database-driven application development in java ee using glassfish;* 1 ed; p. 400; 2008.

22. Object-Relational Mapping. http://en.wikipedia.org/wiki/Object-relational_mapping

23. EHCache http://www.ehcache.org/documentation

24. Hector G. M, Jeffrey D. Ullman, Jennifer W. *Database systems, the complete book*. 2nd Edition. Upper Sandle, NJ: Pearson Education Inc; 2009

APPENDIX A

INSTALLATION GUIDE

This guide assumes that the user already has a copy of the GlycoVault on his local machine. If not available, the user needs to get register with the GlycoVault repository in order to download a fresh copy of GlycoVault. Once the user obtains a fresh copy from the repository, the user needs to have the following prerequisites installed on his/her machine.

Prerequisites

i. PostgreSQL: Download and install PostgreSQL 9.1 from http://www.postgresql.org/download Once you download a copy of Postgres you have to setup a password for the PostgreSQL.

ii. Apache Ant: If not installed already, download and install the ant jar file from http://ant.apache.org/bindownload.cgi**.** Download the latest version and set the ANT_HOME in the Linux path and point to the location of the directory of the downloadable directory

iii. JDK: Download the appropriate Java for your system from the site: http://www.java.com/en/download/manual.jsp.

Once installed, set the environment variable JAVA_HOME to point to the location of the installed version of Java, i.e., the directory containing bin and lib directories.

Say, you installed JDK 7 under /usr/java/ on your Linux machine. Use command

export JAVA_HOME=/usr/java/JDK7

After all the dependent binaries and the libraries have been download and their classpath have been set, the next step is to run the ant script and to populate the database with the initial administrator and user group

APPENDIX B

USER GUIDE

This guide will give a detailed overview of how to use the GlycoVault and what needs to be setup in order to run the GlycoVault.

The first thing that we need is to populate the database with an SQL file. There is a "db" folder in the root directory of GlycoVault, which contains the script called "glycovault.sql" that will generate the required schema and the initial insertions for User, Laboratory and Usergroup table in database.

After the script is executed, the next step is to setup the GlycoVault to run with the database. In order to setup the database in GlycoVault, The user needs to go to the package "edu/uga/cs/glycovault/gvpersist/gvdbaccess/impl" and modify the DbAccessConfig.java file and change the database settings accordingly. After the database settings are done, the initial setting for the GlycoVault is complete. The next step is to log user into the system and create a session for the logged user. The session is created for each user separately and contains the unit of work. Ones the user is logged In, the user can perform different types of operation related to entity such as creating new entities or define association between the entities.

In order to query the entities and their associations based on filters, the user has to use another factory called the FilterFactory. This factory makes sure that the filters related to specific entities and associations are separately created based on the attributes that are defined in the addAttributeFilter method.

One more package that is important in the user guide and which will help the developer in writing the code is the test package, which contains the test cases providing the detailed implementation of all the entities used in the domain object model and how they are associated with each other. The package “edu/uga/cs/glycovault/objectandpersistenttestcases” contains different test cases for different scenarios that occur in GlycoVault. These test cases also provide a guide of how to use the GlycoVault. In order to run these test cases, the JUnit 4.0.jar should be set in the class path.

APPENDIX C

DEVELOPER'S GUIDE

The developer's guide provides all the information of how GlycoVault is working internally. It provides an inside manual to the developers who want to extend the project or change the functionality of the project.

GlycoVault consist of different parts performing various different operations. These parts are describes as follows.

Database Script File

The database creation file "glycovault.sql" is used to initially create a new schema in the Postgres Database server. This file will clean the GlycoVault database schema if there is already exist and also creates the new tables for the system. This file is also used to insert the initial values for the administrator user including the User Group in which the user belongs to and the laboratory in which he works in. These values will make sure that when the GlycoVault runs the first time, the user is able to log into the database. After that the user can use the application to change. The entire database changes needs to be done in this file.

Configuration File.

One more important file, which is required by the GlycoVault, is GlycoVault configuration file named "gvaultconfig.json". This file contains the information about the

Domain Object Model. The file is divided into entity and association information. The following information relates to the entity.

Table 16: The JSON configuration file entity attributes and its description

Name	Provides the name of the entity.
Abstract	Boolean value representing whether the entity class is abstract or not.
Comment	Provides the comment related to the entity.
Attributes	All the attributes related to the entity.
Comments	Provide the comment related to individual attribute.
Types	Provide the data type of the attribute
Getters	The getter method for each attribute of the entity.
Parent	The name of the parent class if exist.

Similarly in the case of the associations between the entities, the following information is provided.

Table 17: The JSON configuration file association attributes and its description

Name	The name of the association
From	The entity from where the association start
To	The entity to which the association is connected
Cardinality	The cardinality for the association e.g. MM.
Ordered	Boolean value indicating whether the association is ordered or not.

This file has various functions. Some of the functionalities for this file are described below.

i. Object and Persistent Layer Generation

One of the purposes of the configuration files is creation of object and the persistent layers. The object layer contains the entity and the association classes. The persistent layer defines the persistence objects for each entity in the domain object model.

ii. SQL statements

The second purpose of the configuration file is the generation of SQL queries (insert, update and select statements) using the ModelSpec class

Code Generators

One of the features of the GlycoVault is that you don't need to manually create the object layer entities and the persistent objects. The code generation feature of GlycoVault will auto generate the files based on the configuration and the template files that are located in the meta folder. The code generators are located in the glycovault/edu/uga/cs/utils package. The information for these files is given in the following table.

Table 18: The code generator java files for GlycoVault

Name	Description
Z_EntityFactory_Gen.java	This file is responsible for generating the entities and the entity factory
Z_EntityImpl_Gen.java	This file is used to create the implementation of the entity files
Z_Factories_Gen.java	This file is responsible for creating the entity factory interface and its implementation class file, association Factory interface and its Implementation class file, the Persistent Factory interface and its Implementation class file.
Z_PersistImpl_Gen.java	This file is responsible for generation of all the implementation class files related to the persistent entities and the associations.
Z_CodeGenerator.java	This file is used to compile and run the above method generator files.

Template Files

In generating the above-mentioned files, GlycoVault also provides the template files that are used to define the way these files should look like when they are generated. The generators read these files before they generate the desired output files. The following template file gives an idea of how the template looks like.

```
/*******************************************************************************
 * @file    $TNAME$ASSOC$UNAME.java
 * @author  Matthew Eavenson, Ki Tae Myoung
 * @version 0.9
 * @date    10/20/2010
 */

package edu.uga.cs.glycovault.gvobject.association;

import java.util.Iterator;

import edu.uga.cs.glycovault.filter.Filter;
import edu.uga.cs.glycovault.gvexcept.GVException;
import edu.uga.cs.glycovault.gvobject.entity.$TNAME;
import edu.uga.cs.glycovault.gvobject.entity.$UNAME;

/*******************************************************************************
 * This interface defines methods for MANY-to-ONE associations.
 * Association type: MANY-to-ONE
 */
public interface $TNAME$ASSOC$UNAME extends AssociationM1<$TNAME, $UNAME>
{
    /*******************************************************************************
     * Get the left side objects of the $TNAME$ASSOC$UNAME object.
     * @param $ULOWER  the right side of the $TNAME$ASSOC$UNAME association.
     * @param $TLOWERFilter  the filter for $TLOWER.
     * @return  the left side ($TNAME) of the association.
     */
    Iterator<$TNAME> get$TNAMEs($UNAME $ULOWER, Filter $TLOWERFilter) throws GVException;

    /*******************************************************************************
     * Get the right side object of the $TNAME$ASSOC$UNAME object.
     * @param $TLOWER  the left side of the $TNAME$ASSOC$UNAME association.
     * @return  the right side ($UNAME) of the association.
     */
    $UNAME get$UNAME($TNAME $TLOWER) throws GVException;

} // $TNAME$ASSOC$UNAME interface
```

Figure 26: Example of template file

This template file is one of the files that are used to create the Interfaces and Classes in GlycoVault. This file is used to create the association interface between two entities. The "$" sign is used to indicate that it will be used by the generator to replace it with the generated code. E.g. TaskIsPartOfExperiment. The other text in the file is used as it is in the Java file.

Generic Packages

The generic packages provide the interfaces and classes that are not related to a specific domain object model. These interfaces and classes define the general methods of how the entities should behave. Some of the generic packages used in GlycoVault are

Table 19: The generic packages used in GlycoVault

Name	Description
Association/Generic	This package defines the generic interfaces used for defining different types of associations and the operations they perform
Association/Impl/Generic	This package provides the implementation classes for the generic association interfaces and implements the methods for those interfaces.
Persistent/Generic	This package defines the interfaces related to the persistent entities and the associations
Persistent/Impl/Generic	This package implements the persistent interfaces for entities and associations and the operations they have.

Db Access Package

This package contains all the classes related to the database. This package makes sure that all the operation that is related to entities and association and in general related to database is performed properly. Some of the important files in this package are

Table 20: The DB access package files and their description

Name	Description
AssociationManagerImpl.java	This file is responsible for handling all the operations related to the association such as updating, creating and deleting them
EntityManagerImpl.java	This file is responsible for handling the operations related to entities.
DbUtils.java	This file is used in interacting with database directly. As in this file all the connections are made, the queries are created and executed and other database related operations
DbAccessConfig.java	This file contains the value related to the database server used and the username and password for the database.
ModelSpec.java	This file is used to read the JSON configuration file and contains functions to retrieve information related to different parts of the JSON files to be used in different packages in GlycoVault i.e. getting the child nodes for a specific entity or getting the association cardinality and other related information.

APPENDIX D

TEST QUERIES AND RESULTS

This appendix provides the GlycoVault schema and details of the queries and that are used for the performance test. As it is mentioned in the comparison chapter, there are 5 different queries used. We will discuss each query and show the SQL and JPQL version of the query. Figure 24 illustrates the UML class diagram for GlycoVault.

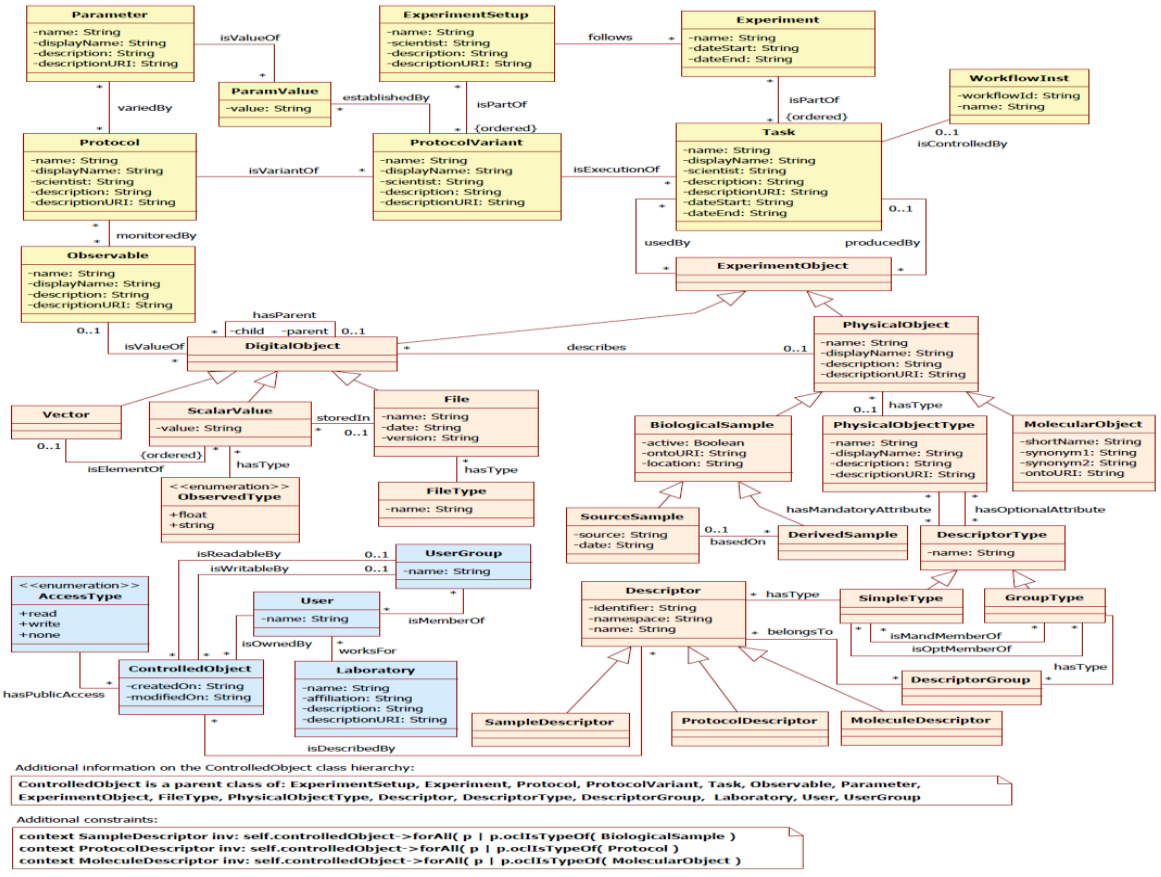


Figure 27. GlycoVault UML Class Diagram[2]

[2] GlycoVault UML class diagram designed by Dr. Krzysztof J. Kochut

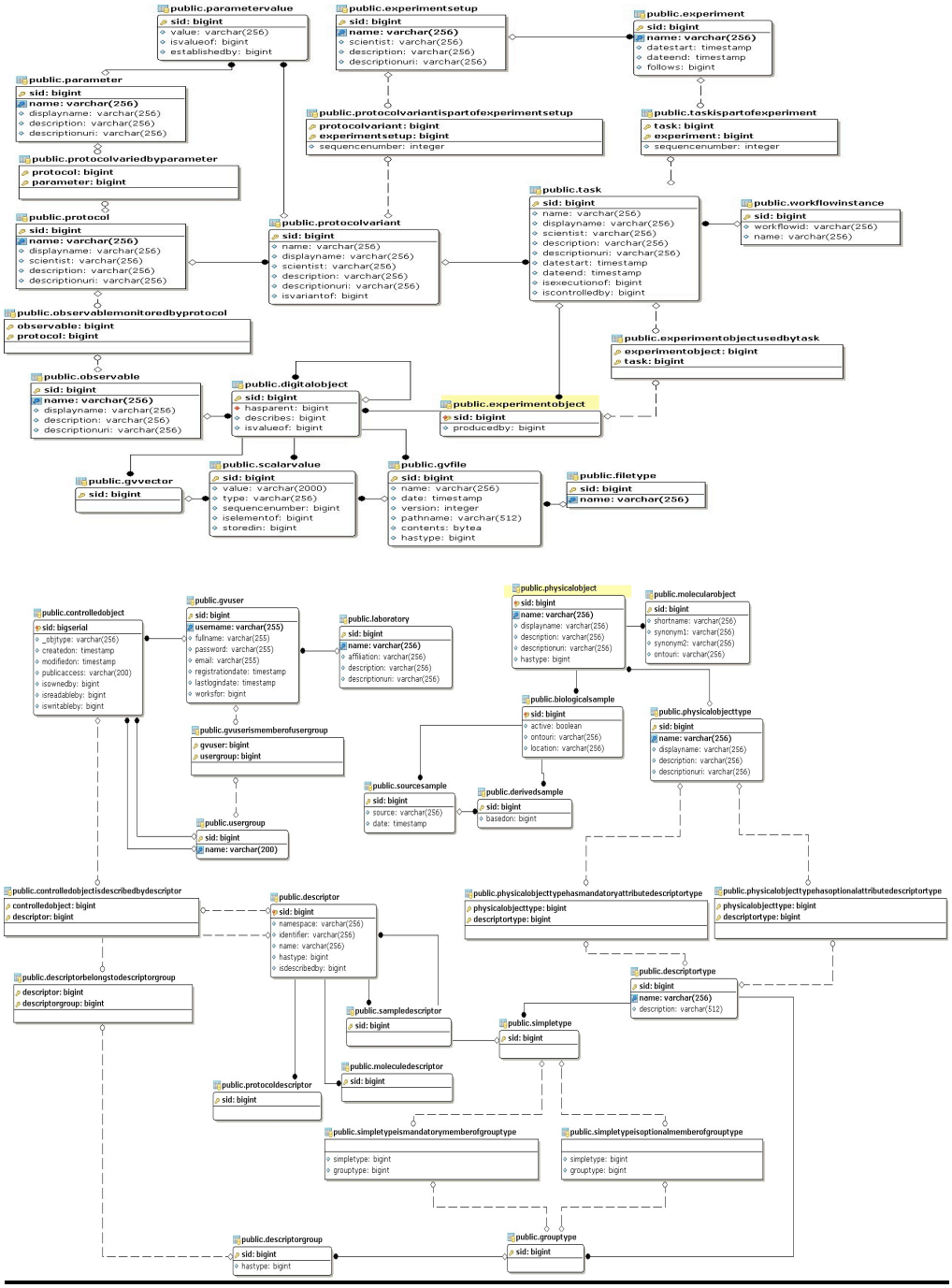


Figure 28. GlycoVault Database Schema Diagram

As we can see, the database schema diagram is split into two parts where the top half describes the experimental part of the GlycoVault and the lower half shows the descriptor and the Authentication part. Both the parts are connected to each other through the ExperimentalObject, which is highlighted

Next we explain the queries used for performance test.

QUERY 1

Given the Experiment Id, get the Experimental data. The SQL query will be as following

The following are the SQL and JPQL queries respectively

```
SELECT Observable.sid as Obserble ,scalarValue.sid, ScalarValue.value, ScalarValue.type,
ControlledObject.createdOn, ControlledObject.modifiedOn,ControlledObject.publicAccess
FROM ScalarValue
NATURAL JOIN ControlledObject
NATURAL JOIN ExperimentObject
NATURAL JOIN DigitalObject
JOIN Observable on observable.sid = digitalobject.isvalueof
JOIN physicalobject on physicalobject.sid = digitalobject.describes
JOIN task on Experimentobject.producedby = task.sid
JOIN taskispartofexperiment tpe on tpe.task = task.sid
JOIN experiment on Experiment.sid = tpe.experiment
WHERE EXISTS
       (SELECT *
       FROM gvuserismemberofusergroup uu
       WHERE (ControlledObject.isOwnedBy = 1)
       OR (ControlledObject.publicAccess != 'none')
       OR (uu.gvuser = 1 AND (uu.userGroup = ControlledObject.isReadableBy
       OR  uu.userGroup = ControlledObject.isWritableBy)))
       and experiment.sid = 2261)
```

```
sv_list = em.createQuery("select ob,sv from DigitalObjectImpl sv"
                        + " JOIN sv.physicalobject ph"
                        + " JOIN sv.observable ob"
                        + " JOIN sv.task t "
                        + " JOIN t.exp tpe"
                        + " where EXISTS (select ug from UserGroupImpl ug"
                        + " join ug.gvuser us_id WHERE sv.isownedby.sid = :uid"
                        + " OR sv.publicAccess != 'none' "
                        + " OR (us_id = 1 AND (ug.sid = sv.isreadableby "
                        + " OR ug.sid = sv.iswritableby))) AND tpe.pkey.experiment.sid = :exp)")
                        .setParameter("uid", user.get(0).getSid())
                        .setParameter("exp", ((ExperimentImpl)exp[1]).getSid())
                        .getResultList();
```

Figure 29: Query 1 in SQL and JPQL

QUERY 2

Given the Experiment Setup, get the Experimental Data.

The following are the SQL and JPQL queries respectively

```
SELECT Observable.sid as Obserble ,scalarValue.sid, ScalarValue.value, ScalarValue.type,
ControlledObject.createdOn, ControlledObject.modifiedOn, ControlledObject.publicAccess
FROM ScalarValue
NATURAL JOIN ControlledObject
NATURAL JOIN ExperimentObject
NATURAL JOIN DigitalObject
JOIN Observable on observable.sid = digitalobject.isvalueof
JOIN physicalobject on physicalobject.sid = digitalobject.describes
JOIN task t1 on Experimentobject.producedby = t1.sid
JOIN taskispartofexperiment te ON te.task = t1.sid
JOIN experiment e on te.experiment = e.sid
JOIN experimentsetup on e.follows = experimentsetup.sid
JOIN taskispartofexperiment tpe on tpe.experiment = e.sid
JOIN Task t2 on tpe.task = t2.sid
JOIN experimentobjectusedbytask et on et.task = t2.sid
JOIN experimentobject eo on eo.sid = et.experimentobject
JOIN physicalobject po on po.sid = eo.sid
JOIN biologicalsample bo on bo.sid = po.sid
JOIN sourcesample ss on ss.sid = bo.sid
WHERE EXISTS
         (SELECT *
         FROM gvuserismemberofusergroup uu
         WHERE (ControlledObject.isOwnedBy = 1)
         OR (ControlledObject.publicAccess != 'none')
         OR (uu.gvuser = 1 AND (uu.userGroup = ControlledObject.isReadableBy
         OR uu.userGroup = ControlledObject.isWritableBy)))
         and ss.sid = 92031
         and experimentsetup.sid = 87544
```

```
sv_list = em.createQuery("select sv from DigitalObjectImpl sv"
                    + " JOIN sv.physicalobject ph"
                    + " JOIN sv.observable ob"
                    + " JOIN sv.task t "
                    + " JOIN t.exp tpe"
                    + " JOIN tpe.pkey.experiment ex"
                    + " where EXISTS (select ug from UserGroupImpl ug"
                    + " join ug.gvuser us_id WHERE sv.isownedby.sid = :uid"
                    + " OR sv.publicAccess != 'none' "
                    + " OR (us_id = 1 AND (ug.sid = sv.isreadableby "
                    + " OR ug.sid = sv.iswritableby))) AND ex.experimentsetup IN (:esetup)")
                    .setParameter("uid", user.get(0).getSid())
                    .setParameter("esetup", esetup)
                    .getResultList();
}
```

Figure 30: Query 2 in SQL and JPQL

QUERY 3

Given the Protocol, get the Experimental Data.

The following are the SQL and JPQL queries respectively

```
SELECT Observable.sid as Obserble ,scalarValue.sid, ScalarValue.value, ScalarValue.type,
ControlledObject.createdOn, ControlledObject.modifiedOn, ControlledObject.publicAccess
FROM ScalarValue
NATURAL JOIN ControlledObject
NATURAL JOIN ExperimentObject
NATURAL JOIN DigitalObject
JOIN Observable on observable.sid = digitalobject.isvalueof
JOIN physicalobject on physicalobject.sid = digitalobject.describes
JOIN task on Experimentobject.producedby = task.sid
JOIN protocolvariant on protocolvariant.sid = task.isexecutionof
JOIN protocol on protocol.sid = protocolvariant.isvariantof
WHERE EXISTS
          (SELECT *
          FROM gvuserismemberofusergroup uu
          WHERE (ControlledObject.isOwnedBy = 1)
          OR (ControlledObject.publicAccess != 'none')
          OR (uu.gvuser = 1 AND (uu.userGroup = ControlledObject.isReadableBy
          OR uu.userGroup = ControlledObject.isWritableBy)))
          and protocol.sid = 2251
```

```
sv_list = em.createQuery("select sv from DigitalObjectImpl sv"
                        + " JOIN sv.physicalobject ph"
                        + " JOIN sv.observable ob"
                        + " JOIN sv.task t "
                        + " JOIN t.protocolvariant pv"
                        + " JOIN pv.protocol prot"
                        + " WHERE EXISTS (select ug from UserGroupImpl ug"
                        + " JOIN ug.gvuser us_id WHERE sv.isownedby.sid = :uid"
                        + " OR sv.publicAccess != 'none' "
                        + " OR (us_id = 1 AND (ug.sid = sv.isreadableby "
                        + " OR ug.sid = sv.iswritableby))) AND prot IN (:prot)")
                        .setParameter("uid", user.get(0).getSid())
                        .setParameter("prot", protlist)
                        .getResultList();
}
```

Figure 31: Query 3 in SQL and JPQL

QUERY 4

Given the Source Sample, get the Experimental Data

The following are the SQL and JPQL queries respectively

```
SELECT Observable.sid as Obserble ,scalarValue.sid, ScalarValue.value, ScalarValue.type,
ControlledObject.createdOn, ControlledObject.modifiedOn, ControlledObject.publicAccess
FROM ScalarValue
NATURAL JOIN ControlledObject
NATURAL JOIN ExperimentObject
NATURAL JOIN DigitalObject
JOIN Observable on observable.sid = digitalobject.isvalueof
JOIN physicalobject on physicalobject.sid = digitalobject.describes
JOIN task t1 on Experimentobject.producedby = t1.sid
JOIN taskispartofexperiment te ON te.task = t1.sid
JOIN experiment e on te.experiment = e.sid
JOIN taskispartofexperiment tpe on tpe.experiment = e.sid
JOIN Task t2 on tpe.task = t2.sid
JOIN experimentobjectusedbytask et on et.task = t2.sid
JOIN experimentobject eo on eo.sid = et.experimentobject
JOIN physicalobject po on po.sid = eo.sid
JOIN biologicalsample bo on bo.sid = po.sid
JOIN sourcesample ss on ss.sid = bo.sid
WHERE EXISTS
          (SELECT *
          FROM gvuserismemberofusergroup uu
          WHERE (ControlledObject.isOwnedBy = 1)
          OR (ControlledObject.publicAccess != 'none')
          OR (uu.gvuser = 1 AND (uu.userGroup = ControlledObject.isReadableBy
          OR uu.userGroup = ControlledObject.isWritableBy)))
          and ss.sid = 92031
```

```
sv_list = em.createQuery("select sv from DigitalObjectImpl sv"
                        + " JOIN sv.physicalobject ph"
                        + " JOIN sv.observable ob"
                        + " JOIN sv.task t"
                        + " JOIN t.exp te"
                        + " JOIN te.pkey.experiment e "
                        + " JOIN e.expt tep"
                        + " JOIN tep.pkey.task t2"
                        + " WHERE EXISTS (SELECT eb from t2.uexperimentobject eb WHERE eb.sid = (:sd)) "
                        + " AND EXISTS (select ug from UserGroupImpl ug"
                        + " JOIN ug.gvuser us_id WHERE sv.isownedby.sid = :uid"
                        + " OR sv.publicAccess != 'none' "
                        + " OR (us_id = 1 AND (ug.sid = sv.isreadableby "
                        + " OR ug.sid = sv.iswritableby)))" )
                        .setParameter("uid", user.get(0).getSid())
                        .setParameter("sd", new Long(2266))
                       .getResultList();
}
```

Figure 32: Query 4 in SQL and JPQL

QUERY 5

Given the Experiment Setup and Source Sample, get the Experimental Data.

The following are the SQL and JPQL queries respectively

```
SELECT Observable.sid as Obserble ,scalarValue.sid, ScalarValue.value, ScalarValue.type,
ControlledObject.createdOn, ControlledObject.modifiedOn, ControlledObject.publicAccess
FROM ScalarValue
NATURAL JOIN ControlledObject
NATURAL JOIN ExperimentObject
NATURAL JOIN DigitalObject
JOIN Observable on observable.sid = digitalobject.isvalueof
JOIN physicalobject on physicalobject.sid = digitalobject.describes
JOIN task t1 on Experimentobject.producedby = t1.sid
JOIN taskispartofexperiment te ON te.task = t1.sid
JOIN experiment e on te.experiment = e.sid
JOIN experimentsetup on e.follows = experimentsetup.sid
JOIN taskispartofexperiment tpe on tpe.experiment = e.sid
JOIN Task t2 on tpe.task = t2.sid
JOIN experimentobjectusedbytask et on et.task = t2.sid
JOIN experimentobject eo on eo.sid = et.experimentobject
JOIN physicalobject po on po.sid = eo.sid
JOIN biologicalsample bo on bo.sid = po.sid
JOIN sourcesample ss on ss.sid = bo.sid
WHERE EXISTS
        (SELECT *
        FROM gvuserismemberofusergroup uu
        WHERE (ControlledObject.isOwnedBy = 1)
        OR (ControlledObject.publicAccess != 'none')
        OR (uu.gvuser = 1 AND (uu.userGroup = ControlledObject.isReadableBy
        OR uu.userGroup = ControlledObject.isWritableBy)))
        and ss.sid = 92031
        and experimentsetup.sid = 87544
```

```
List<ScalarValue>sv_list2 = em.createQuery("select sv from DigitalObjectImpl sv"
                        + " JOIN sv.physicalobject ph"
                        + " JOIN sv.observable ob"
                        + " JOIN sv.task t"
                        + " JOIN t.exp tpe"
                        + " JOIN tpe.pkey.experiment ex"
                        + " JOIN ex.expt tpe2"
                        + " WHERE EXISTS "
                        + "(SELECT eb from tpe2.pkey.task.uexperimentobject eb "
                        + " WHERE eb.sid = (:sd)) AND EXISTS (select ug from UserGroupImpl ug"
                        + " JOIN ug.gvuser us_id WHERE sv.isownedby.sid = :uid"
                        + " OR sv.publicAccess != 'none' "
                        + " OR (us_id = 1 AND (ug.sid = sv.isreadableby "
                        + " OR ug.sid = sv.iswritableby))) AND ex.experimentsetup.sid = :eid")
                        .setParameter("uid", user.get(0).getSid())
                        .setParameter("sd", new Long(2266))
                        .setParameter("eid", new Long(2261))
                       .getResultList();
}
```

Figure 33: Query 5 in SQL and JPQL

Table 21: Table containing the averages of queries on different tools for 5000 records (time in ms)

Tools / Queries	Glycovault	Hibernate	EclipseLink	OpenJPA	Ebean
Query 1	771.81	895.68	2469.25	3100.22	1087.79
Query 2	933.38	1319.09	2503.55	3352.23	1190.59
Query 3	830.80	991.47	2500.11	3280.11	1178.37
Query 4	1238.87	1632.23	2632.54	3682.20	1511.87
Query 5	1710.61	1880.02	2852.88	3834.42	1896.41

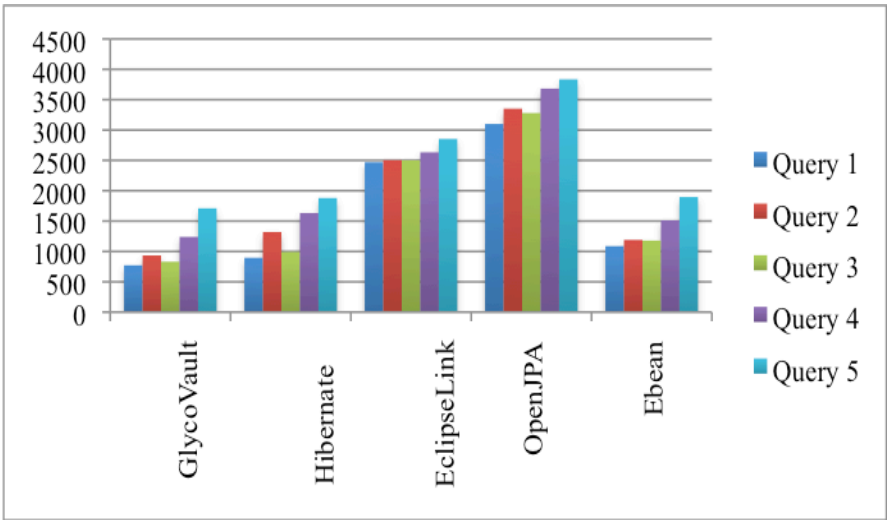


Figure 34: Graph showing averages for 5000 records for persistent tools (time in ms)

Table 22: Table containing the standard deviations of queries on different tools for 5000 records (time in ms)

Tools / Queries	Glycovault	Hibernate	EclipseLink	OpenJPA	Ebean
Query 1	1343.16	1976.75	4681.01	4234.23	4681.01
Query 2	1449.03	2016.90	4733.83	4002.32	4733.83
Query 3	1880.66	2176.31	4789.90	3933.45	4789.90
Query 4	1498.84	2301.47	4790.91	4322.32	4790.91
Query 5	1911.05	1280.02	5173.15	4624.23	5173.15

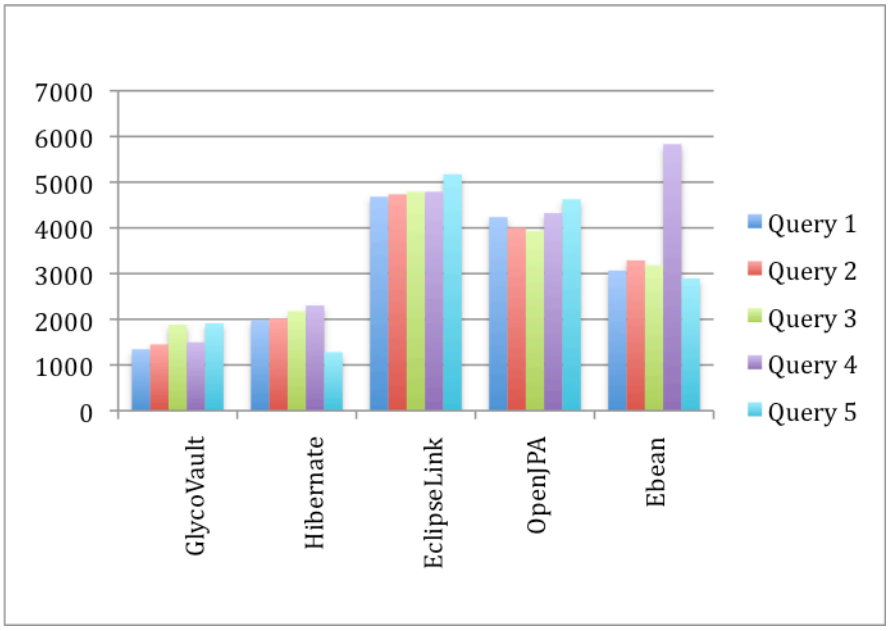


Figure 35: Graph showing standard deviation for 5000 records for persistent tools (time in ms)

Table 23: Table containing the averages of queries on different tools for 10000 records (time in ms)

Tools / Queries	Glycovault	Hibernate	EclipseLink	OpenJPA	Ebean
Query 1	1023.54	1673.49	3700.27	4367.39	1375.23
Query 2	1202.53	1703.84	4023.64	4194.99	1833.86
Query 3	1132.33	1656.10	3934.72	4158.85	1544.59
Query 4	1632.34	1842.24	4290.16	4308.34	1834.34
Query 5	1939.23	2064.68	4523.99	5251.58	1994.89

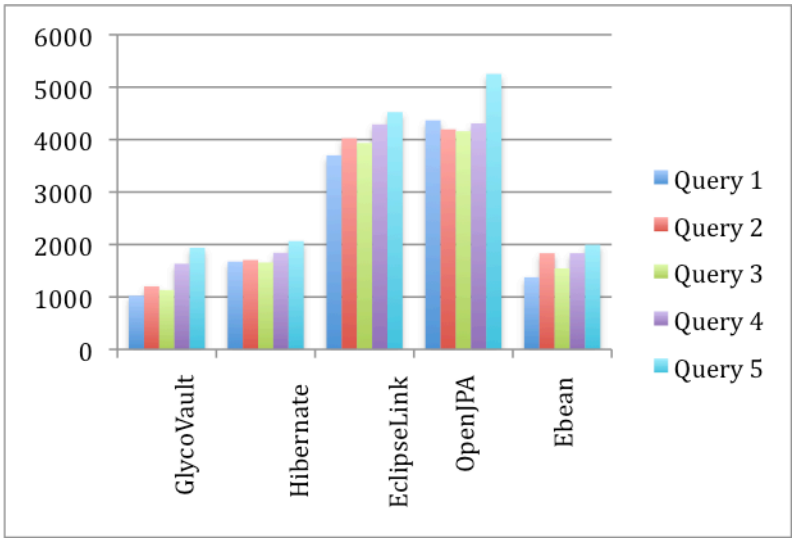


Figure 36: Graph showing averages for 10000 records for persistent tools (time in ms)

Table 24. Table containing the standard deviation of queries on different tools for 10000 records (time in ms)

Tools / Queries	Glycovault	Hibernate	EclipseLink	OpenJPA	Ebean
Query 1	1807.87	2412.50	7679.21	5342.23	4309.95
Query 2	1475.94	2403.94	7550.61	5532.84	4480.61
Query 3	2310.35	2422.75	7564.70	5232.45	4388.08
Query 4	2216.28	2385.27	7640.27	5433.85	7111.23
Query 5	2437.51	2704.79	7412.83	5523.24	4693.04

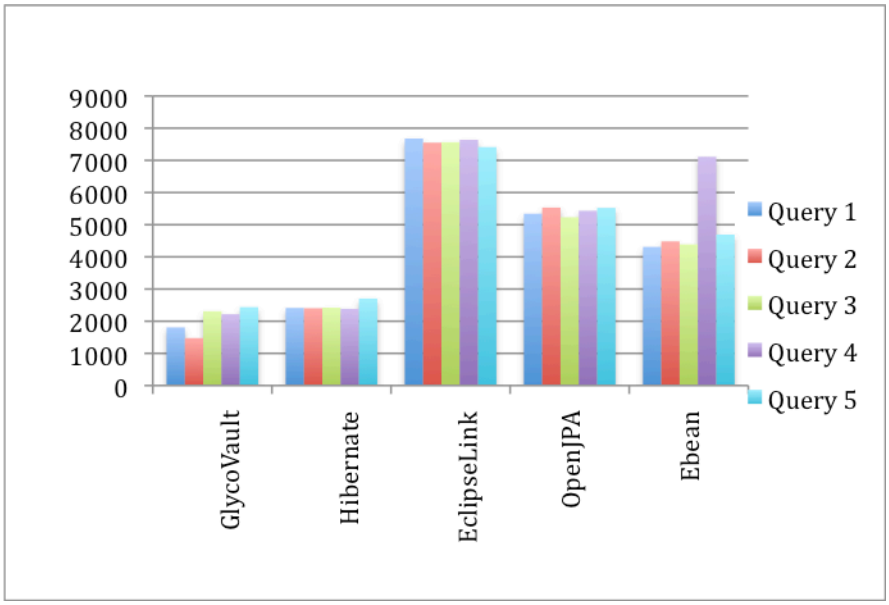


Figure 37. Graph showing standard deviation for 10000 records for persistent tools (time in ms)

Table 25: Table containing the averages of queries on different tools for 20000 records (time in ms)

Tools / Queries	Glycovault	Hibernate	EclipseLink	OpenJPA	Ebean
Query 1	1822.22	2540.33	4904.34	6234.32	3120.23
Query 2	2194.12	2642.43	5290.45	6123.12	3461.17
Query 3	1935.43	2504.32	4942.23	6523.12	3288.09
Query 4	2323.11	3364.02	5505.23	7132.23	3659.13
Query 5	2745.65	3936.65	5767.34	7803.12	3940.49

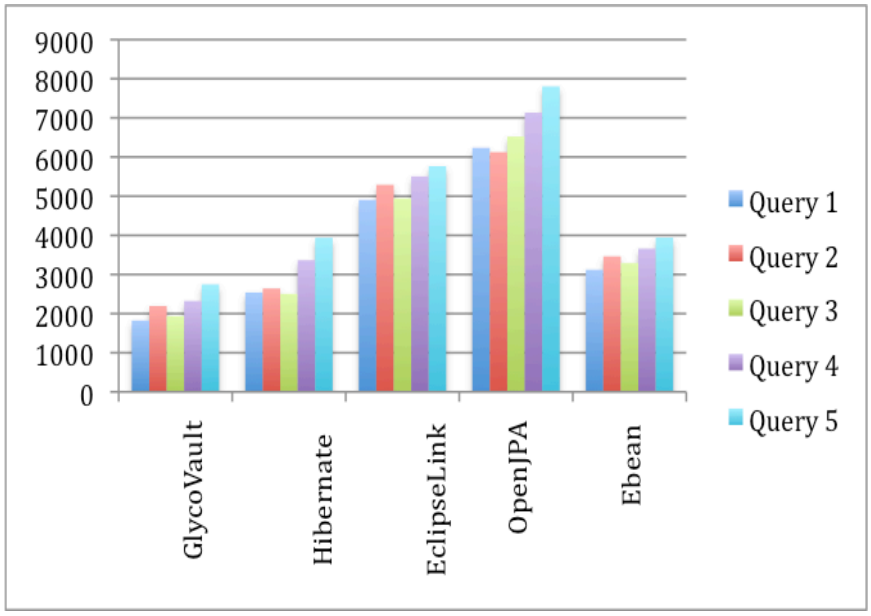


Figure 38: Graph showing averages for 20000 records for persistent tools (time in ms)

Table 26: Table containing the standard deviation of queries on different tools for 20000 records (time in ms)

Tools / Queries	Glycovault	Hibernate	EclipseLink	OpenJPA	Ebean
Query 1	2200.64	3280.26	8945.34	7533.34	3733.62
Query 2	2521.65	3741.70	8544.43	7659.34	3711.41
Query 3	2145.65	3159.45	8521.54	7823.32	3630.49
Query 4	2351.45	3090.42	8334.34	6990.23	3857.64
Query 5	2415.54	3512.13	8225.23	6839.13	3689.62

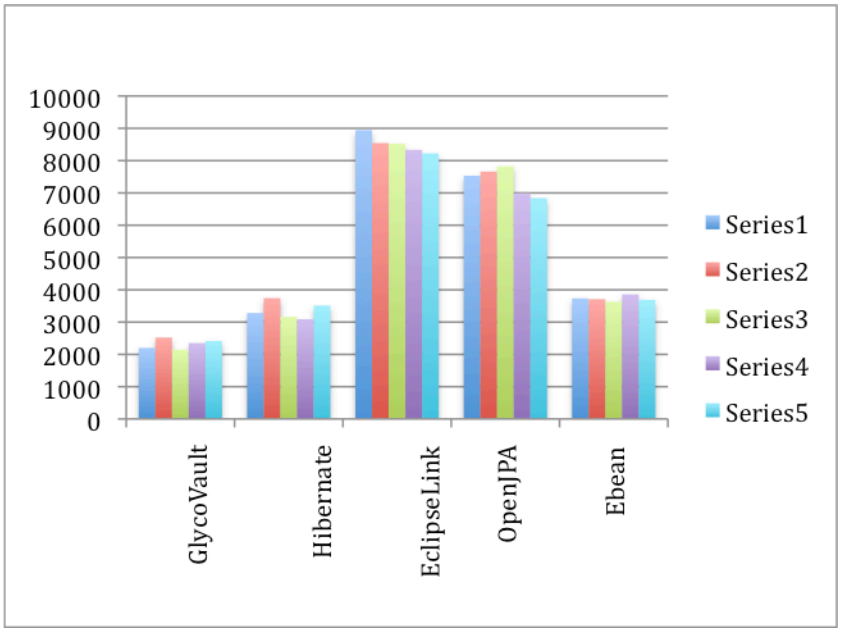


Figure 39: Graph showing standard deviation for 20000 records for persistent tools (time in ms)

Table 27: Table containing the averages of queries on different tools for 40000 records (time in ms)

Tools / Queries	Glycovault	Hibernate	EclipseLink	OpenJPA	Ebean
Query 1	6883.97	7999.53	12013.22	15980.23	5736.78
Query 2	10601.29	9643.59	14340.23	17934.32	5884.24
Query 3	9437.22	8781.05	12322.92	16406.34	5709.12
Query 4	10583.15	10320.32	19234.23	20222.89	8374.75
Query 5	13031.32	12690.18	22233.23	24238.24	10768.06

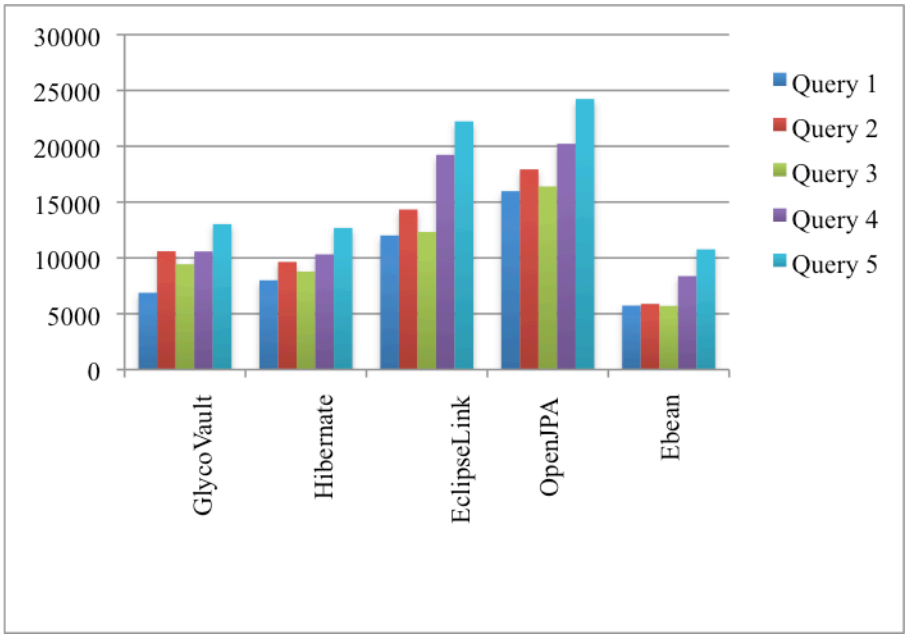


Figure 40: Graph showing averages for 40000 records for persistent tools (time in ms)

Table 28: Table containing the standard deviations of queries on different tools for 40000 records (time in ms)

Tools / Queries	Glycovault	Hibernate	EclipseLink	OpenJPA	Ebean
Query 1	1895.26	42203.21	72235.23	64023.34	6515.35
Query 2	1932.23	41719.21	69933.23	68434.34	6682.23
Query 3	1253.21	43049.78	65235.25	66345.23	6500.34
Query 4	1499.64	39423.32	64469.23	68344.24	23809.11
Query 5	1531.23	43799.62	69434.34	66006.45	6544.29

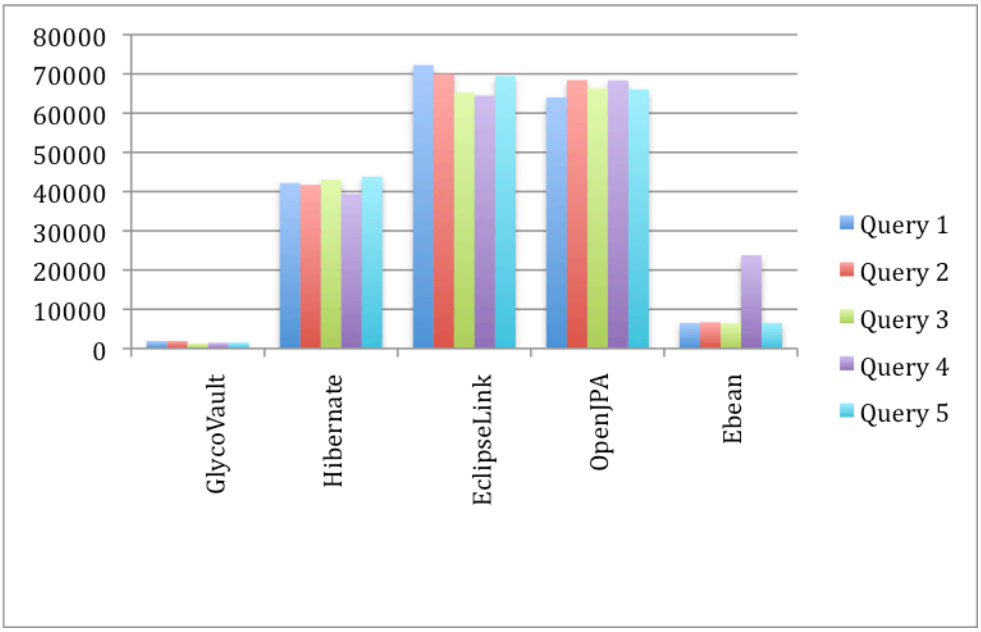


Figure 41: Graph showing standard deviation for 40000 records for persistent tools (time in ms)

Table 29: Table containing the averages of queries on different tools for 80000 records (time in ms)

Tools / Queries	Glycovault	Hibernate	EclipseLink	OpenJPA	Ebean
Query 1	15161.07	20802.24	25233.23	41034.23	16327.47
Query 2	18172.87	23070.23	29343.23	44567.54	18203.32
Query 3	17381.00	21331.21	27990.23	43436.42	17270.31
Query 4	22942.24	25731.25	38932.23	49432.45	20423.35
Query 5	26313.12	28132.23	43999.23	56467.42	24114.23

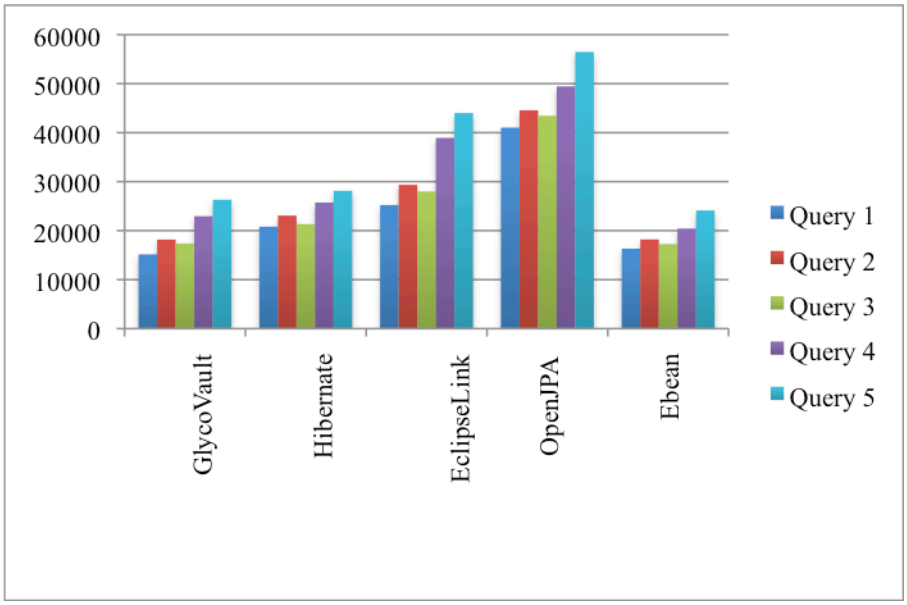


Figure 42: Graph showing averages for 80000 records for persistent tools (time in ms)

Table 30. Table containing the standard deviation of queries on different tools for 80000 records (time in ms)

Tools / Queries	Glycovault	Hibernate	EclipseLink	OpenJPA	Ebean
Query 1	1752.57	55787.23	82342.32	79433.32	13843.30
Query 2	1823.23	52386.24	89902.23	79999.34	13701.42
Query 3	2136.86	52442.85	87234.25	81043.23	13756.24
Query 4	2214.12	52877.23	89224.23	84233.25	53726.34
Query 5	2112.23	3745.23	88023.23	85932.25	3565.23

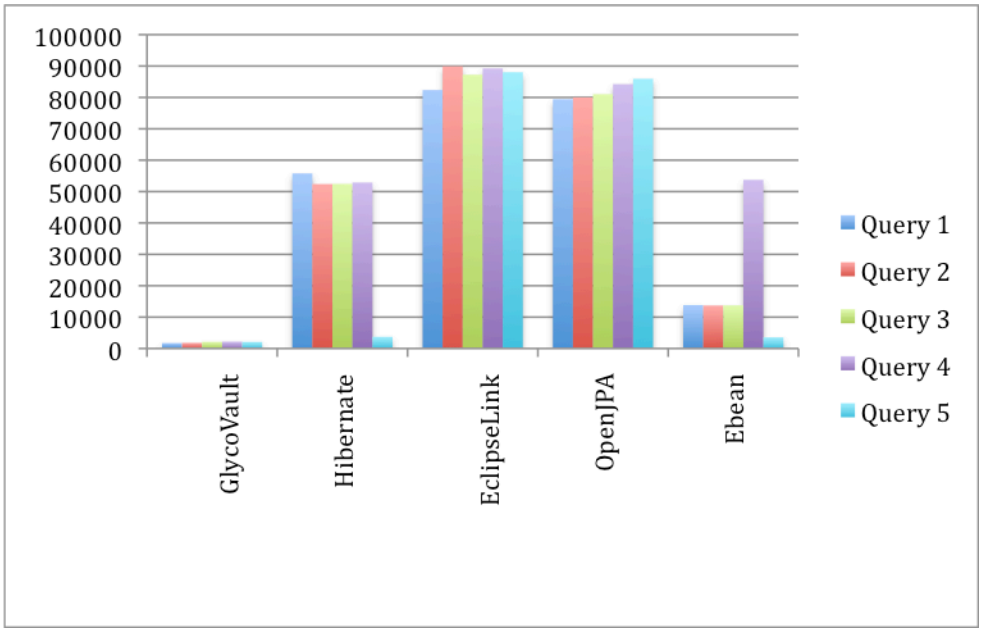


Figure 43: Graph showing standard deviation for 80000 records for persistent tools (time in ms)

Table 31: Table containing the averages of queries on different tools for 160000 records (time in ms)

Tools / Queries	Glycovault	Hibernate	EclipseLink	OpenJPA	Ebean
Query 1	24170.82	30980.41	33233.23	69423.21	24715.23
Query 2	28290.27	34402.52	38234.63	71340.35	24842.25
Query 3	27124.23	31254.25	35642.43	70932.32	24492.32
Query 4	32123.14	36347.98	44346.36	78232.23	32650.24
Query 5	34987.78	40224.25	51998.34	83323.23	34772.25

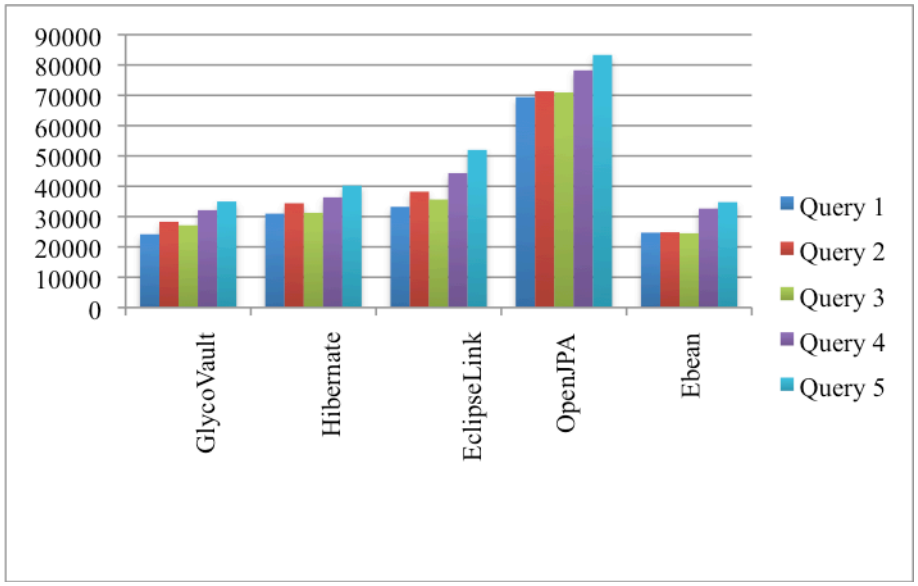


Figure 44: Graph showing averages for 160000 records for persistent tools (time in ms)

Table 32: Table containing the standard deviation of queries on different tools for 160000 records (time in ms)

Tools / Queries	Glycovault	Hibernate	EclipseLink	OpenJPA	Ebean
Query 1	2012.25	55370.35	86435.34	88434.23	27693.34
Query 2	2002.23	54236.22	87353.34	85434.23	27818.23
Query 3	2432.23	55746.23	86435.34	89343.23	27435.54
Query 4	2012.14	55491.33	92234.25	87324.23	24520.26
Query 5	2220.87	55342.25	91342.45	88233.32	27754.79

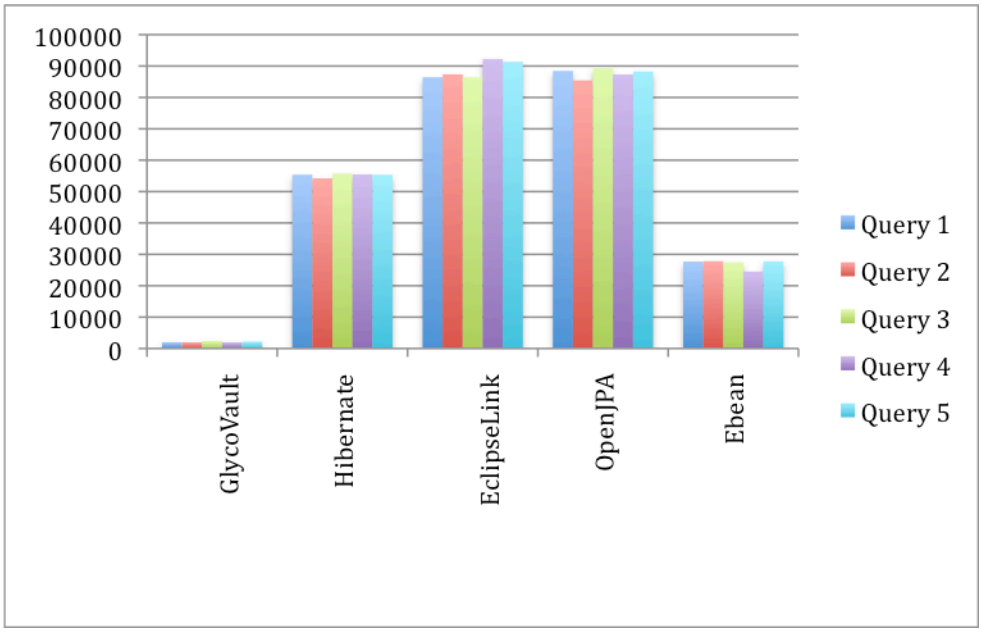


Figure 45: Graph showing standard deviation for 160000 records for persistent tools (time in ms)

Table 33: Table containing the averages of queries on different tools for 5000 records with cache (time in ms)

Tools / Queries	Glycovault	Hibernate	EclipseLink	OpenJPA	Ebean
Query 1	376.06	533.34	826.24	868.12	568.65
Query 2	556.67	683.24	944.53	949.45	597.12
Query 3	376.96	553.92	889.98	840.23	559.58
Query 4	696.40	769.25	962.34	1038.23	735.63
Query 5	821.34	899.57	1044.12	1219.23	953.72

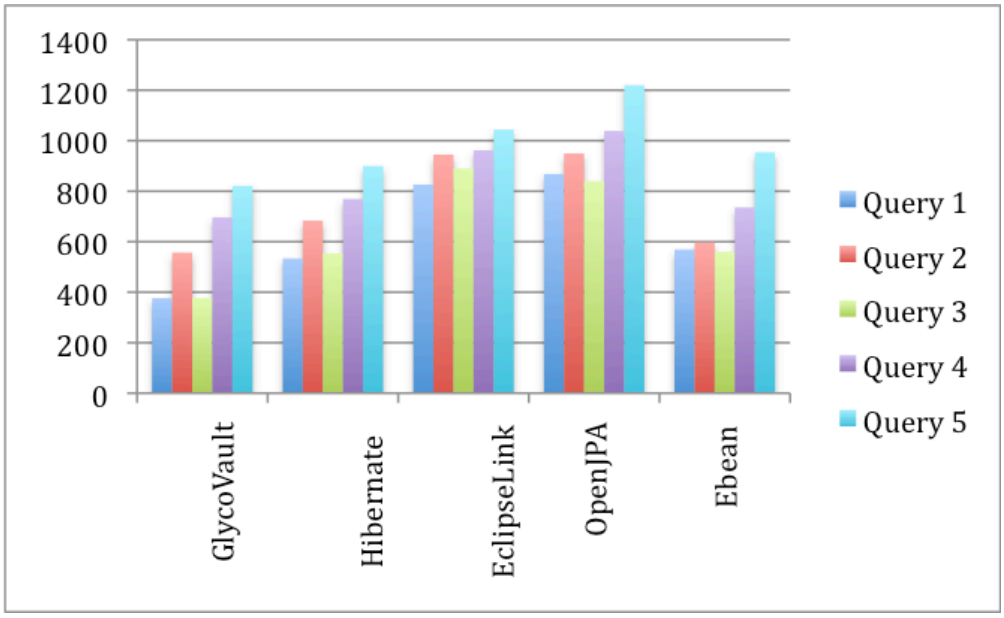


Figure 46: Graph showing averages for 5000 records for persistent tools with cache (time in ms)

Table 34: Table containing the standard deviations of queries on different tools for 5000 records with cache (time in ms)

Tools / Queries	Glycovault	Hibernate	EclipseLink	OpenJPA	Ebean
Query 1	597.46	599.12	1034.23	843.03	2512.55
Query 2	561.06	653.45	1131.35	884.30	2163.46
Query 3	740.45	794.32	1423.21	760.34	2145.52
Query 4	629.61	887.34	1332.34	1093.34	3336.87
Query 5	753.45	988.34	1387.25	1232.25	3631.76

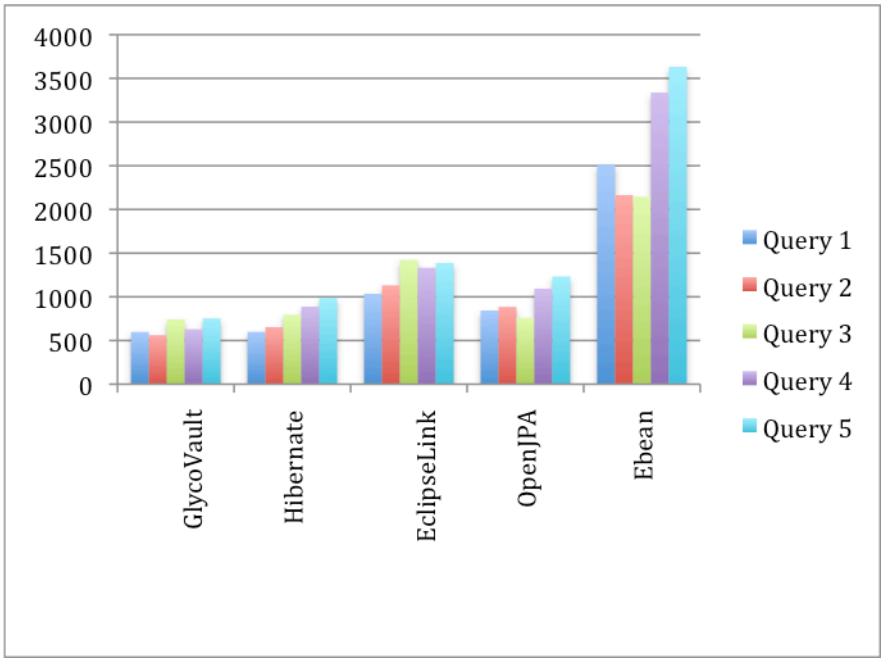


Figure 47: Graph showing standard deviation for 5000 records for persistent tools with cache (time in ms)

Table 35. Table containing the averages of queries on different tools for 10000 records with cache (time in ms)

Tools / Queries	Glycovault	Hibernate	EclipseLink	OpenJPA	Ebean
Query 1	627.94	853.32	1143.45	1192.54	883.64
Query 2	982.34	989.98	1354.43	1484.34	1023.50
Query 3	882.65	770.57	1084.43	1245.43	961.35
Query 4	950.28	889.29	1196.53	1776.34	1252.95
Query 5	1120.43	993.32	1229.32	1960.23	1493.33

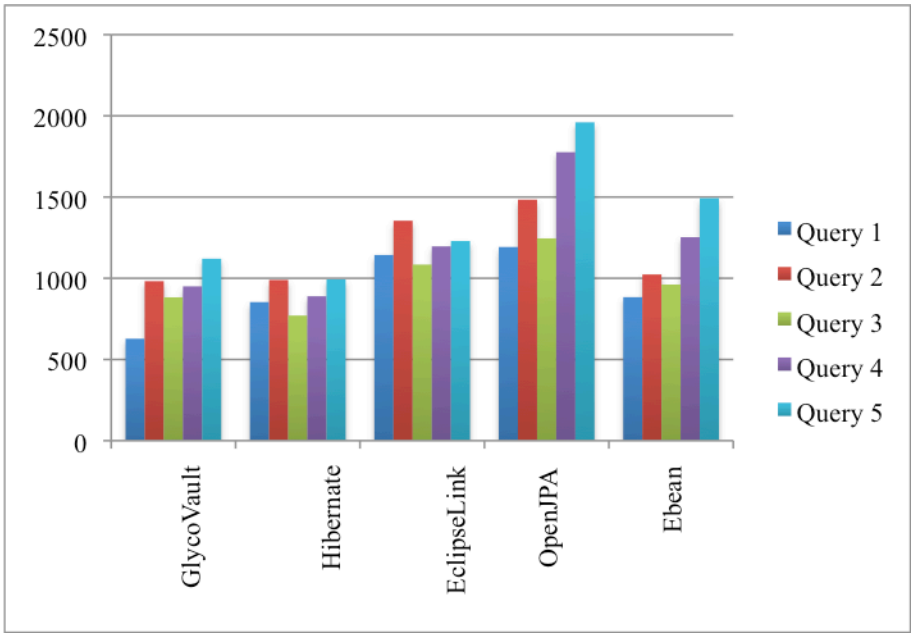


Figure 48: Graph showing averages for 10000 records for persistent tools with cache (time in ms)

Table 36. Table containing the standard deviations of queries on different tools for 10000 records with cache (time in ms)

Tools / Queries	Glycovault	Hibernate	EclipseLink	OpenJPA	Ebean
Query 1	1022.24	998.34	1133.34	1029.40	3204.44
Query 2	1172.05	1009.32	1443.24	1129.02	3056.17
Query 3	1393.86	1092.43	1332.54	1019.34	2901.04
Query 4	1212.79	1223.23	1543.23	998.12	5312.64
Query 5	1321.52	1022.34	1454.43	872.01	2845.06

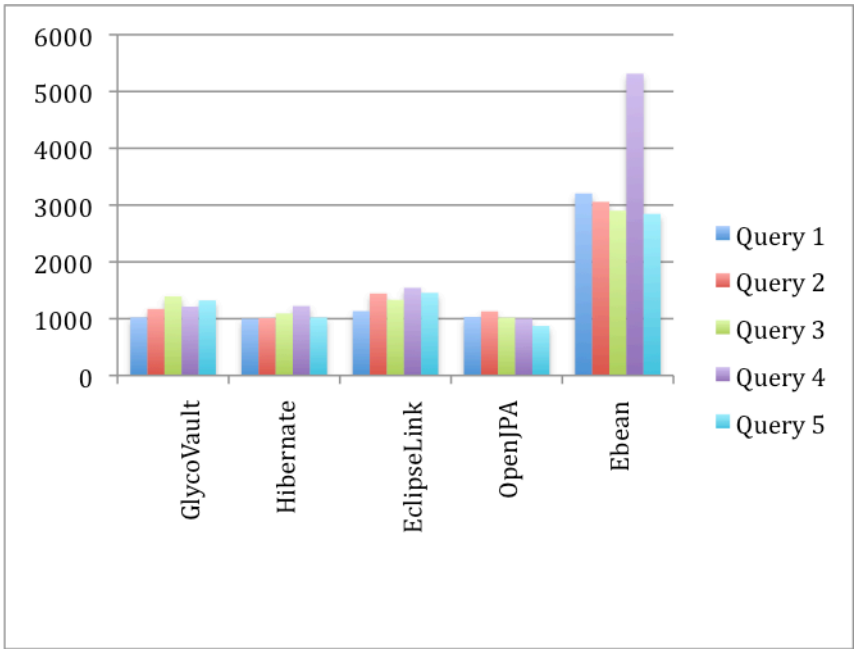


Figure 49: Graph showing standard deviation for 10000 records for persistent tools with cache (time in ms)

Table 37: Table containing the averages of queries on different tools for 20000 records with cache (time in ms)

Tools / Queries	Glycovault	Hibernate	EclipseLink	OpenJPA	Ebean
Query 1	1298.94	923.24	1591.32	2041.34	1168.81
Query 2	1505.26	1245.32	1851.34	2398.34	1427.27
Query 3	1359.57	1021.23	1669.23	2104.43	1308.52
Query 4	1853.05	1349.25	2398.14	2434.32	1828.25
Query 5	2234.24	1453.54	2783.54	2742.34	2147.37

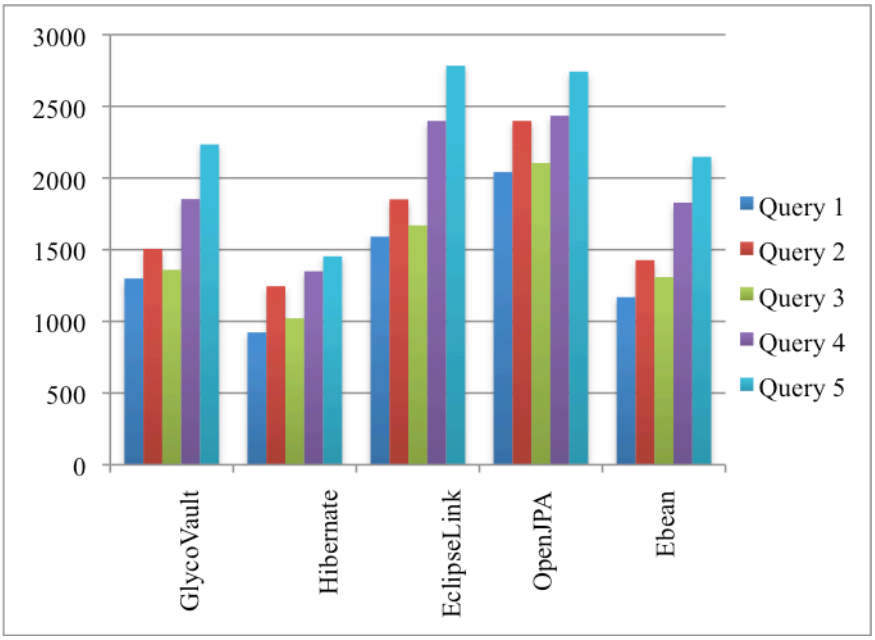


Figure 50: Graph showing averages for 20000 records for persistent tools with cache (time in ms)

Table 38: Table containing the standard deviations of queries on different tools for 20000 records with cache (time in ms)

Tools / Queries	Glycovault	Hibernate	EclipseLink	OpenJPA	Ebean
Query 1	1559.96	1229.23	1864.34	1001.23	4419.84
Query 2	1528.58	1132.32	1787.23	1293.50	4315.84
Query 3	2121.21	1533.14	1880.34	1323.25	4269.23
Query 4	1636.72	1694.43	1768.23	1198.98	9551.62
Query 5	3244.23	1553.20	1988.23	1443.98	4363.41

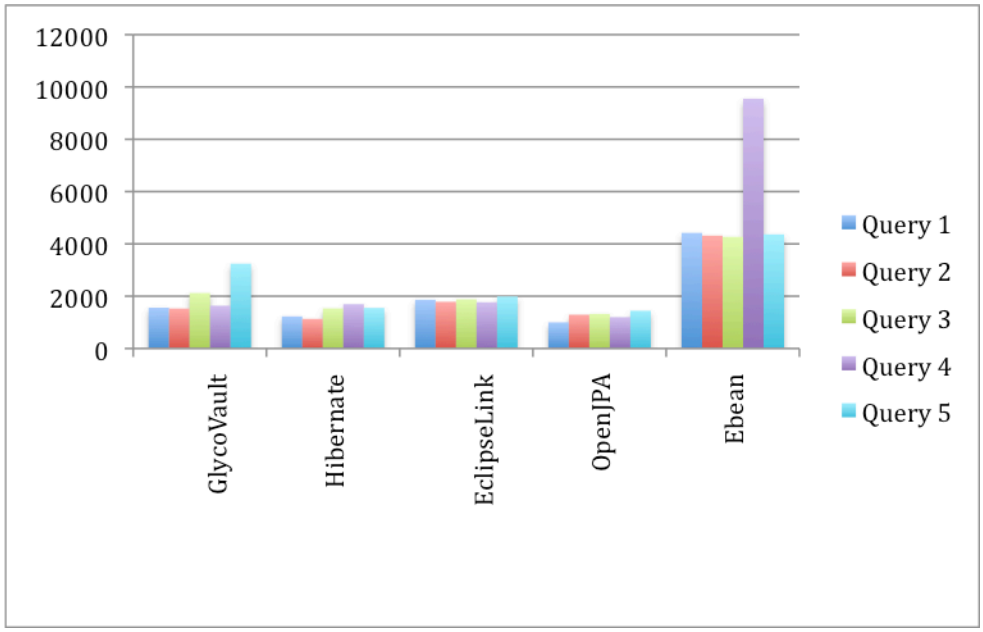


Figure 51: Graph showing standard deviation for 20000 records for persistent tools with cache (time in ms)

Table 39: Table containing the averages of queries on different tools for 40000 records with cache (time in ms)

Tools / Queries	Glycovault	Hibernate	EclipseLink	OpenJPA	Ebean
Query 1	2732.25	1238.32	2253.23	3694.98	2270.22
Query 2	3231.23	1595.32	2663.23	4076.69	2830.16
Query 3	3024.89	1323.23	2483.23	3598.86	2113.02
Query 4	3734.02	1644.03	2973.23	4743.45	2804.23
Query 5	3955.04	1765.03	3432.23	5286.79	3247.54

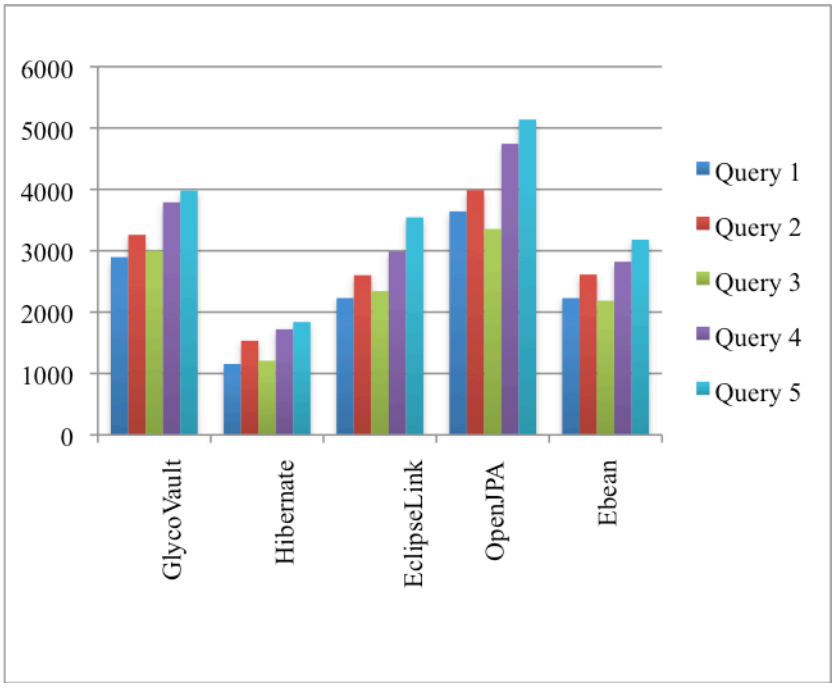


Figure 52: Graph showing averages for 40000 records for persistent tools with cache (time in ms)

Table 40: Table containing the standard deviations of queries on different tools for 40000 records with cache (time in ms)

Tools / Queries	Glycovault	Hibernate	EclipseLink	OpenJPA	Ebean
Query 1	1432.25	1898.32	1753.23	1344.98	9070.22
Query 2	1331.23	1995.32	1663.23	1276.69	6530.16
Query 3	1524.89	1823.23	1983.23	1198.86	6713.02
Query 4	1534.02	1944.03	1873.23	1543.45	5904.23
Query 5	1855.04	1765.03	1932.23	1186.79	7947.54

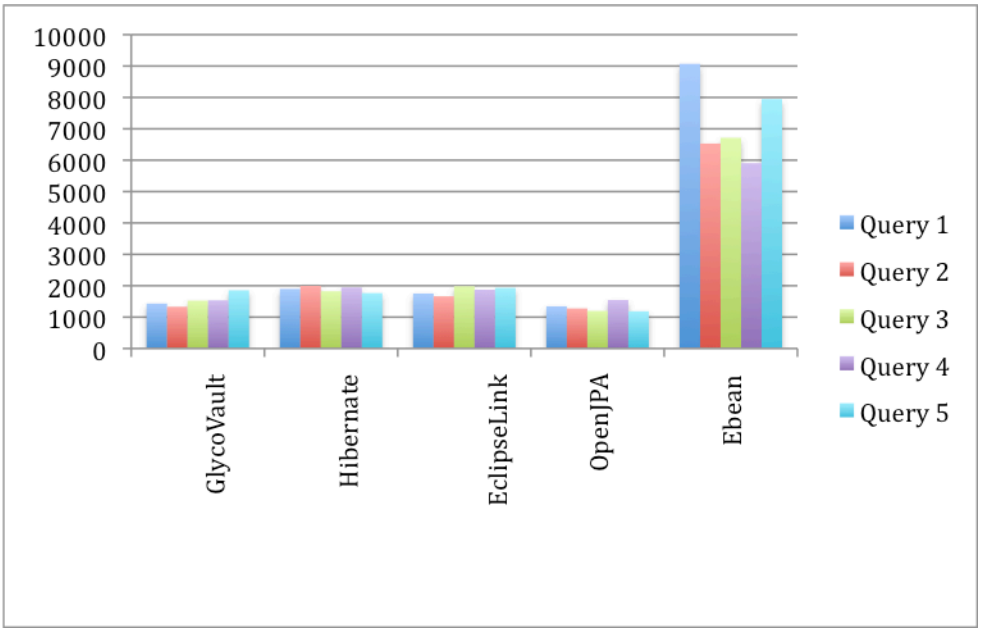


Figure 53: Graph showing standard deviation for 40000 records for persistent tools with cache (time in ms)

Table 41: Table containing the averages of queries on different tools for 80000 records with cache (time in ms)

Tools / Queries	Glycovault	Hibernate	EclipseLink	OpenJPA	Ebean
Query 1	4355.69	1806.23	2992.12	5686.32	2986.97
Query 2	4685.95	2187.79	3490.23	5972.44	3643.23
Query 3	4204.29	1977.98	3270.43	5480.46	3272.34
Query 4	5054.56	2765.93	3990.23	6440.23	4323.23
Query 5	5632.23	2990.45	4430.2	7987.21	4804.43

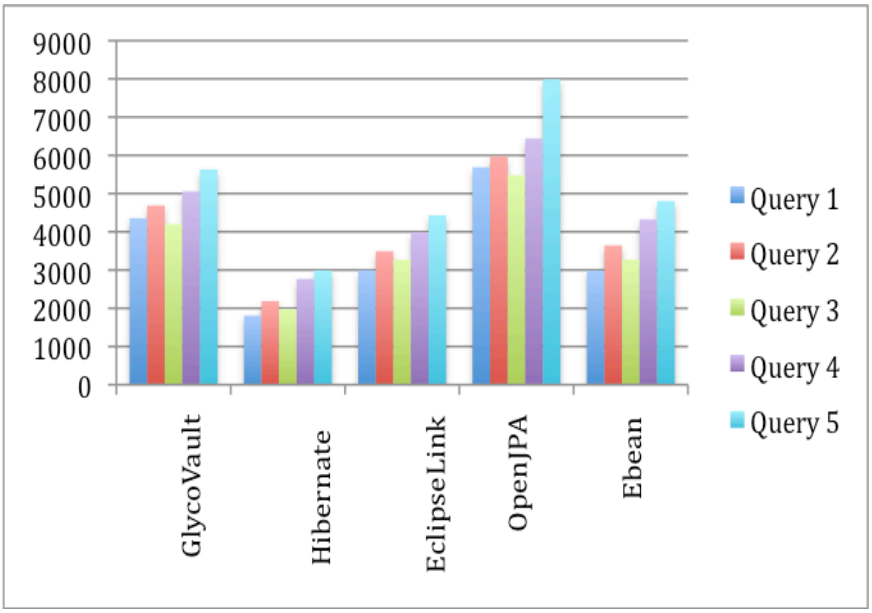


Figure 54: Graph showing averages for 80000 records for persistent tools with cache (time in ms)

Table 42: Table containing the standard deviations of queries on different tools for 80000 records with cache (time in ms)

Tools / Queries	Glycovault	Hibernate	EclipseLink	OpenJPA	Ebean
Query 1	1642.49	2948.02	2214.23	1982.23	10023.21
Query 2	1772.34	2948.25	2003.23	1873.32	9985.25
Query 3	1983.23	2874.34	1993.23	1569.08	9787.23
Query 4	1839.03	3009.23	2201.42	1685.23	11998.23
Query 5	1933.56	2787.20	2553.02	2210.23	9980.25

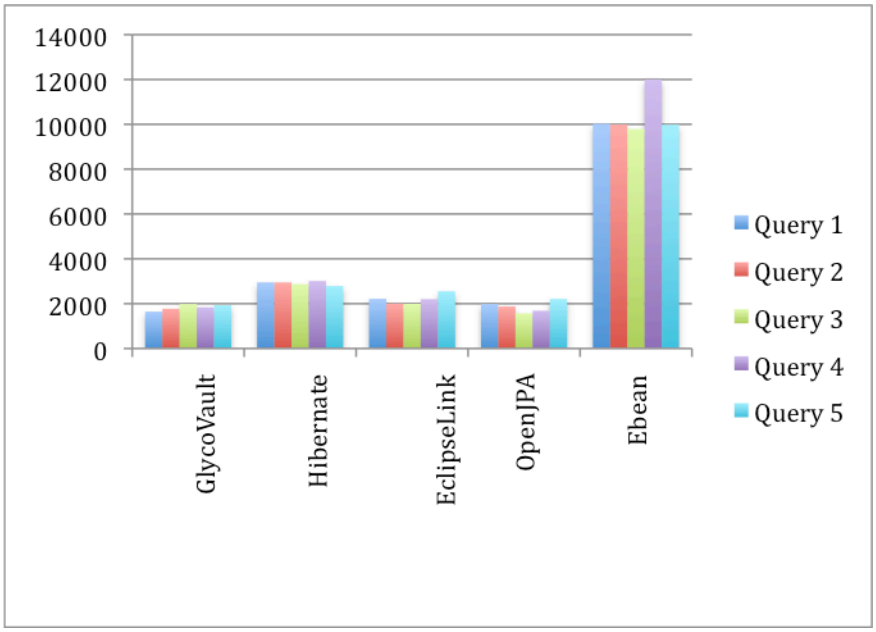


Figure 55: Graph showing standard deviation for 80000 records for persistent tools with cache (time in ms)

Table 43: Table containing the averages of queries on different tools for 160000 records with cache (time in ms)

Tools / Queries	Glycovault	Hibernate	EclipseLink	OpenJPA	Ebean
Query 1	7696.05	2890.97	5523.32	7324.23	5686.98
Query 2	9671.70	3275.34	5743.12	7953.34	4997.96
Query 3	8423.98	2901.98	5210.32	7267.23	6198.87
Query 4	10557.91	3889.89	6370.54	8222.32	6432.23
Query 5	11434.25	4196.87	6933.12	8465.45	5686.98

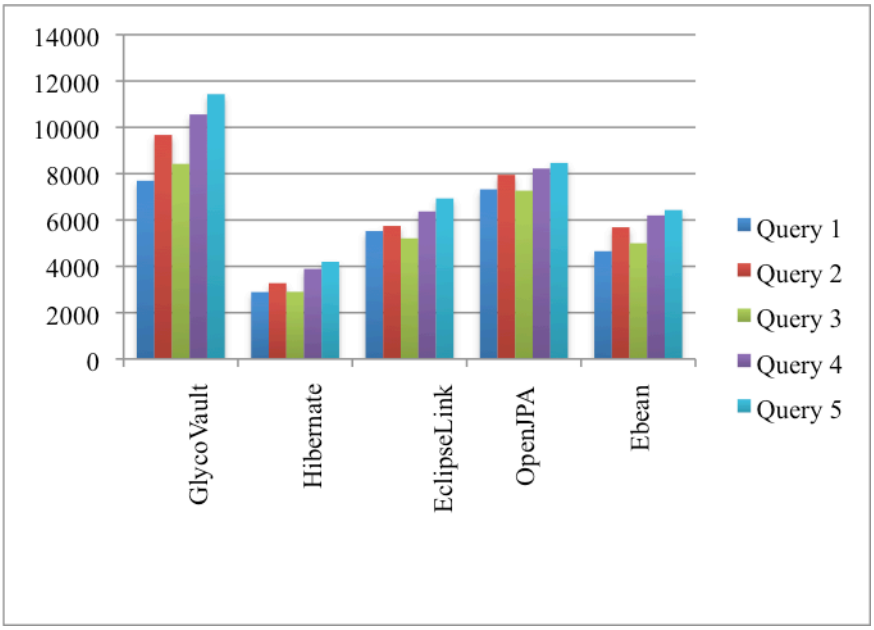


Figure 56: Graph showing averages for 160000 records for persistent tools with cache (time in ms)

Table 44: Table containing the standard deviations of queries on different tools for 160000 records with cache (time in ms)

Tools / Queries	Glycovault	Hibernate	EclipseLink	OpenJPA	Ebean
Query 1	1255.23	5523.23	2302.24	2022.34	9902.32
Query 2	892.23	5343.50	3222.32	2103.23	10032.23
Query 3	516.27	5519.20	2533.02	2294.27	10334.36
Query 4	1566.85	5221.04	2290.19	1932.09	11230.32
Query 5	992.25	5323.23	2109.0	1889.23	1009.24

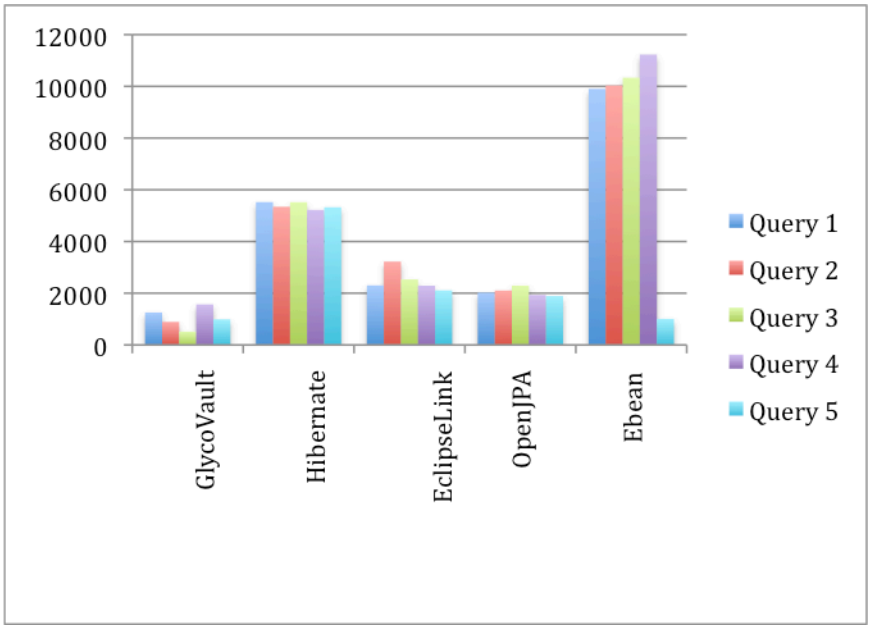


Figure 57: Graph showing standard deviation for 160000 records for persistent tools with cache (time in ms)

APPENDIX E

POPULATION OF GLYCOMICS DATA (USE CASE)

One of the main purpose of developing GlycoVault is to help the bioinformatics' scientists and the researchers store the scientific data and to retrieve it in different ways. In this appendix we will show a brief overview of how the actual data are being extracted and then used by GlycoVault to get the experimental data.

GlycoVault instantiates objects called Isobolic sets, which are defined by references to GlycO ontology where the individual structures that comprise the isobolic sets are enumerated. GlycoVault describes the abundance of each isobolic set in each sample. For given sample retrieve the transcript data and glycomics data. Glycomics data consist of isobolic sets and their abundance. For each isobolic set, find the glycan structure in GlycO and for each structure, find reactions in ReactO ontology. For each reaction, find enzymes (enumerated in EnzyO ontology) that catalyze under the constraint that enzymes should be from the same organism, which produces the sample.

There are several steps involved in getting the experimental data based on the Glycans structure. These steps are described below.

Steps

1. Start with a specific reaction in ReactO.
2. Find the reactants, which are specified as molecules consumed by the reaction.
3. Find the products, which are specified as molecules generated by the reaction.

4. Find the isobolic sets containing the molecules under the constraint that the isobolic set was used to describe the glycomics analysis. This infers that the structures are present in the organism that generated the sample.
5. Find the abundance of each isobolic set in the sample, which is in the GlycoVault.
6. Find the enzyme(s) that catalyze the reaction under the constraints that the same organism that generated the sample produces these enzymes.
7. Find the genes that encode the enzymes in that organism.
8. Find the transcript abundance for each of these genes in GlycoVault for this biological sample.

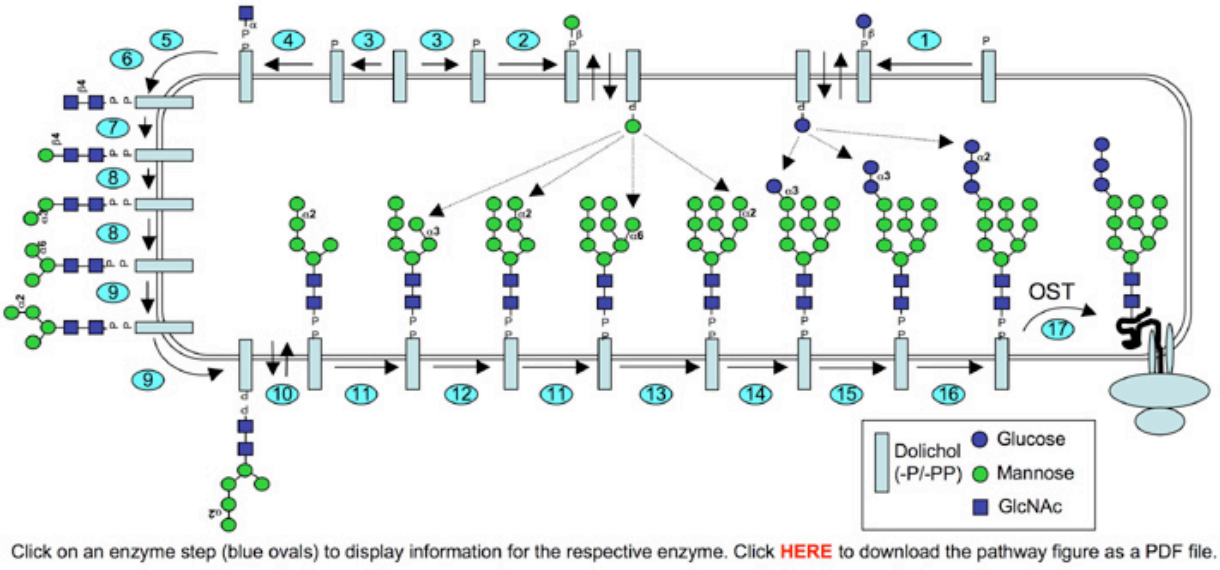


Figure 58: Showing the Glycans Pathway

As shown in the figure 58 above the blue oval shaped reactions gets the reactants as the glycans structures and based on the reactions it will produce the desired Glycan structure. One of the sample data XML files that are used to create the glycans structure is shown below.

```
<?xml version="1.0" encoding="UTF-8"?>
<spectral_segment_list xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <originator>Mike Tiemeyer</originator>
  <date>2012-04-02</date>
  <source name="E:\Research\Data\Data_from_Meng\new-O-glycan-list.xml" original_source_name=
  <data_source name="E:\Research\Data\Data_from_Meng\Pro5-standard\Pro5-O-standard-rep3-trap
  <series number="1" time="0"/>
  <structure derivative="Me" end_structure="alditol" minZ="1" maxZ="4">
    <adduct name="Na" min="1" max="4"/>
  </structure>
  <isomer name="(Hex)1(HexNAc)1" type="O-linked oligosaccharide">
    <residue_list>
      <residue abbrev="Hex" number="1"/>
      <residue abbrev="HexNAc" number="1"/>
    </residue_list>
    <elemental_composition>
      <atom name="H" number="45"/>
      <atom name="C" number="23"/>
      <atom name="N" number="1"/>
      <atom name="O" number="11"/>
    </elemental_composition>
    <subspectrum peaksCount="2242" scan="5" lowMZ="533.1892" highMZ="538.395" basePeakMZ="53
      <ion_structure charge="1" monoisotopicMZ="534.2892065750001">
        <charge_carrier name="Na" number="1"/>
      </ion_structure>
      <peaks precision="32" byteOrder="network" pairOrder="m/z-int">NTMzLjE4OTIJMC4wCjUzMy4x
    <peak_width>0.008322984374718 98</peak_width><delta>0.002739841406550114</delta><statisti
    <subspectrum>
      <ion_structure charge="2" monoisotopicMZ="278.63960328750005">
        <charge_carrier name="Na" number="2"/>
      </ion_structure>
    </subspectrum>
    <subspectrum>
      <ion_structure charge="3" monoisotopicMZ="193.42306885833338">
        <charge_carrier name="Na" number="3"/>
      </ion_structure>
    </subspectrum>
    <subspectrum>
      <ion_structure charge="4" monoisotopicMZ="150.81480164375003">
        <charge_carrier name="Na" number="4"/>
```

Figure 59: XML data file for the O-Glycans

This file contains the information about the Isobols and the O –Glycans. The tags in this file provide a lot of information about the scalar values and the observables and the Isobols that they describe. There are 26 observables used in this file. These observables are as follows

Table 45. Observable names from the O-Glycan XML files

Observables
1. Mass to Charge (m/z)
2. Mass (m)
3. Charge_carrier (z)
4. Charge_ion
5. Peak_width
6. delta
7. slope_mono
8. slope_intercept
9. slope_purged
10. slope_all
11. intercept_mono
I2. Ntercept_intercept
13. intercept_purged
I4. intercept_all
15. rho_mono
16. rho_intercept
17. rho_purged
18. rho_all
19. rho_mono
20.std_error _intercept
21. std_error _purged
22. std_error _all
23. ionCount_mono
24. onCount_intercept
25. onCount_purged
26. onCount_all

Each of these observables have scalar values, which describe the isobols.

APPENDIX F

Example of Object Relational Mapping in Hibernate

This appendix gives an overview of how the persistent classes are created in Hibernate. Also it will guide in how the class metadata are defined for mapping between the classes and the table. We will take an example of Office and Employee class. And we associate two instances of Employee i.e. worker and manager to the Office instance.

```
@Entity
public class Office {
    @Id
    @GeneratedValue
    private long id;

    private String title;

    @OneToOne(fetch = FetchType.LAZY)
    private Employee manager;

    @OneToOne(fetch = FetchType.LAZY)
    private Employee worker;

    public Office() {}

    public Office(Employee m, Employee w, String t)
    {
        manager = m;
        worker = w;
        title = t;
    }
    public Employee getManager() {
        return manager;
    }
    public void setManager(Employee manager) {
        this.manager = manager;
    }
    public Employee getWorker() {
        return worker;
    }
    public void setWorker(Employee worker) {
        this.worker = worker;
    }
    public long getId() {
        return id;
    }
    public void setId(long id) {
        this.id = id;
    }
    public String getTitle() {
        return title;
    }

    public void setTitle(String title) {
        this.title = title;
```

```
    }
  }

@Entity
public class Employee {
    @Id
    @GeneratedValue
    private Long id;

    private String firstName;
    private String lastName;
    @Temporal(javax.persistence.TemporalType.DATE)
    private Date birthDate;

    private String cellPhone;

    public Employee() {}

    public Employee(String fn, String ln, Date bd, String cp)
    {
        firstName = fn;
        lastName = ln;
        birthDate = bd;
        cellPhone = cp;
    }
    public Date getBirthDate() {
        return birthDate;
    }
    public void setBirthDate(Date birthDate) {
        this.birthDate = birthDate;
    }
    public String getCellPhone() {
        return cellPhone;
    }
    public void setCellPhone(String cellPhone) {
        this.cellPhone = cellPhone;
    }
    public String getFirstName() {
        return firstName;
    }
    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }
    public Long getId() {
        return id;
    }
    public void setId(Long id) {
        this.id = id;
    }
    public String getLastName() {
        return lastName;
    }
    public void setLastName(String lastName) {
        this.lastName = lastName;
    }
}
```

Figure 60. Example of Hibernate Annotated Mapped File

This code snippet is for creating an Office class. The @Entity annotation is used to indicate that its an entity class. The @id and @GeneratedValue is used to represent the primary key field and to provide a mechanism for auto generation of keys. @OneToOne annotation provides association with the Employee class instances i.e. worker and manager. Similarly the Employee class provides the information for creating the Employee instance.

```
<persistence-unit name="Thesis_example">
    <provider>org.hibernate.ejb.HibernatePersistence</provider>
    <class>Employee</class>
    <class>Office</class>
    <properties>
      <property name="hibernate.dialect"
value="org.hibernate.dialect.PostgreSQLDialect"/>
      <property name="hibernate.default_schema" value="public"/>
      <property name="hibernate.connection.username"
value="postgres"/>
      <property name="hibernate.connection.driver_class"
value="org.postgresql.Driver"/>
      <property name="hibernate.connection.password"
value="postgres"/>
      <property name="hibernate.connection.url"
value="jdbc:postgresql://localhost:5432/thesis_test"/>
      <property name="hibernate.hbm2ddl.auto" value="update"/>
    </properties>
  </persistence-unit>
```

Figure 61. Example of Persistence.xml file

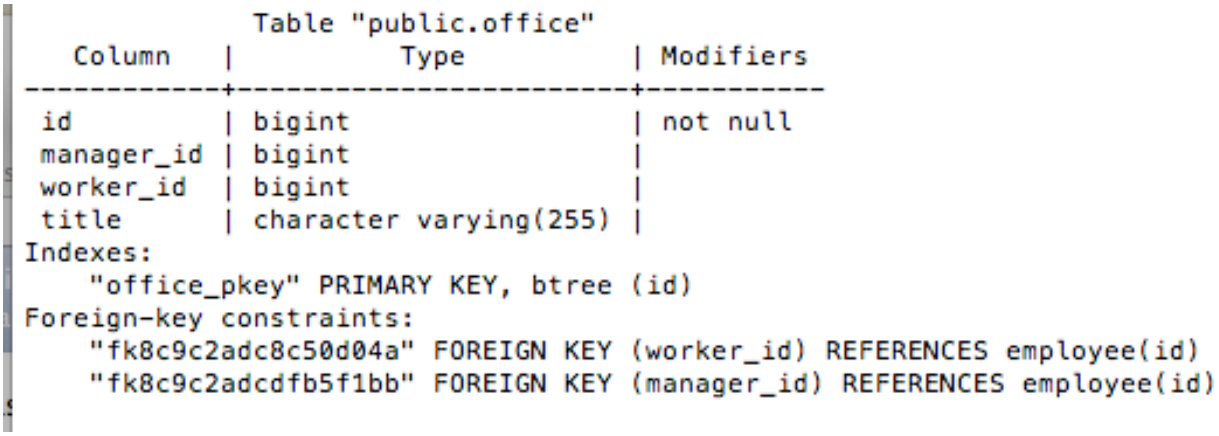
The persistence.xml file shown above contains the information about the classes that needs to be mapped to the database tables. Also, it contains the information about the database connection and login information. Also the user can specify other properties for logging and caching using this file.

```
public class Test_office {
    private static EntityManagerFactory emf;
    private static EntityManager em;
    public static void main(String [] args)
    {
        emf =
Persistence.createEntityManagerFactory("Thesis_example");
```

```
        em = emf.createEntityManager();
        em.getTransaction().begin();
        Employee manager = new Employee("John", "Smith", new Date(),
"123");
        Employee worker = new Employee("Bob", "Smith", new Date(),
"321");
        em.persist(manager);
        em.persist(worker);
        Office office1 = new Office(manager, worker, "computer
science");
        em.persist(office1);
        em.getTransaction().commit();
    } // main
} // Test_Office
```

Figure 62. Example of Hibernate Annotated Mapped File

This is the test file to create the persistent objects. As we can see that once we create an instance of EntityManager class, we can use its methods to provide functionality such as `persist()` and create a new transaction object using `getTransaction()` which can be used for committing a transaction.



```
              Table "public.office"
   Column    |          Type          | Modifiers
-------------+------------------------+-----------
 id          | bigint                 | not null
 manager_id  | bigint                 |
 worker_id   | bigint                 |
 title       | character varying(255) |
Indexes:
    "office_pkey" PRIMARY KEY, btree (id)
Foreign-key constraints:
    "fk8c9c2adc8c50d04a" FOREIGN KEY (worker_id) REFERENCES employee(id)
    "fk8c9c2adcdfb5f1bb" FOREIGN KEY (manager_id) REFERENCES employee(id)
```

```
thesis_test=# select * from office;
 id | manager_id | worker_id |      title
----+------------+-----------+------------------
 19 |         17 |        18 | computer science
(1 row)
```

Figure 63. Records generated by Hibernate for Office and Employee table.

.