Bachelor Thesis Project

Comparison of Java Persistence Layer Technologies

Author: Ang Sun
Supervisor: Johan Hagelbäck
Semester: VT 2016
Subject: Computer Science

Abstract

As data and data structures grown more complex in computing, the task of storing and accessing such data efficiently also becomes more complex. Object-oriented programming languages such as Java have popularized the practice of using class-based objects and created the challenge of persisting these complex objects to storage systems such as databases which only store simple scalar values. This paper seeks to explore and compare a selected number of popular persistence solutions for the Java language in their use and performance. This is done through researching, implementing and performance testing the chosen candidates. Through these steps we have found that while more abstracted solutions provided easier implementation and usage, these positives come with the disadvantages of generally slower performance. We also discovered that while many persistence frameworks provide features that minimize the amount of code required for implementation, they suffer from performance issues such as the N+1 query issue if not configured or utilized correctly.

Contents

1 Introduction

Linking software applications to data stores has grown more complex with time. Increasingly elaborate data models mean that reading and writing information to and from storage solutions such as databases may often not be a trivial task[1].

This paper will provide an overview of persistence layer technologies and a comparison of current Java-based implementations. A theoretical comparison will first be conducted on the functional aspects of the software candidates followed by an experiment to determine how well the software candidates perform in a variety of situations.

1.1 Background

This section will provide explanation and an overview of concepts relevant to the topic of this paper.

1.1.1 Persistence and Databases

Persistence is the concept of allowing data to persist even beyond the lifespan of the process that created it. Typically this is done by writing data to a physical medium which retains information such as a hard-disk drive. On the software level, persistence can be achieved in many ways. Data can be written as raw data files or arranged in more organised manners through additional software layers [1]. One variety of persistent data storage is database technology.

Databases refer to an organised collection of data arranged in a manner so that they may be accessed efficiently and precisely at a later time [2]. They typically consist of schemas, tables, views, stored procedures and more. Modern databases can be divided into relational databases and non-relational databases commonly referred to as NoSQL. Relational databases store data within table columns that have strictly defined constraints. NoSQL spans a variety of database architectures that forgo the typical tabular relations in favour of simpler data structures that vary depending on implementation. Examples include key-value stores, document stores and graph databases. Database management systems (DBMS) provide an interface through which databases may be defined, created, queried, updated and administrated. While databases are generally not portable across DBMSs, standards such as SQL allow for interoperability between applications and multiple DBMS and database systems.

Table 1.1 shows how a user object may be stored in a relational database table. The table consists of multiple columns each containing a specific type of data e.g. text strings and integers. Each row in the table represents one user object and in this case is identified by a unique key `id`.

id	username	password
1	john	abcdefg
2	marie	password
3	ellen	1234
4	mark	pwd123

Table 1.1: A simple relation represent a user in a system.

1.1.2 SQL

SQL or Structured Query Language is a programming language for managing data within a relational database [3]. SQL allows for data and schemas to be inserted, updated, retrieved and deleted. Specific implementations of SQL from various database system vendors may not fully follow standards and therefore be incompatible between implementations. This can cause issues in applications which may have to interact with various databases and thus be database agnostic. SQL can also be extended with features to allow for procedural style programming, common dialects of SQL include PL/SQL and T-SQL (or Transact-SQL)[3].

The SELECT statement is the most commonly used operation in SQL. It queries the database to retrieve data from any given number of tables and returns a list of entries. SELECT statement does not make any alterations to data within a database and is considered to be a read-only action. Assuming a database contains a relation "user", we can use the SQL statement shown in table 1.1 to retrieve specific columns from said relation and filter the data.

```
SELECT u.id, u.username
FROM user u
WHERE u.id = 10;
```

Figure 1.1: Example of SQL SELECT statement

This statement will return the columns id and username from table user for a row that has id 10.

Additional statements include the INSERT statement used for inserting data into a relation; the UPDATE statement for updating existing data in a relation; DELETE statement for removing data in a relation; ALTER statement for making alterations to relations among other statements that may vary from implementation to implementation.

1.1.3 Java and Persistence

Object oriented programming languages such as Java emphasize the usage of class based objects [4]. When the need of persisting these objects arises however, their potential complexity mean that storage using a database becomes difficult since databases can only store simple scalar values [2]. Persistence layer software can help bridge the incompatibility between objects and database storage by automating mapping of SQL queries and actions and/or automatically converting objects into a group of simpler values that can be stored in tables [1].

The `User` class show in Figure 1.2 can be used to represent the relation represented by Figure 1.1. POJO stands for plain old Java object and is a term used to describe Java classes that consist of various fields and getter/setter methods for these fields. Fields are typically encapsulated for data validation and security.

1.1.4 JPA

JPA or Java Persistence API is a specification that dictates how data from a relation data source can be managed using the Java programming language [5]. Some

```
public class User{
    private int id;
    private String username;

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public String getUsername() {
        return username;
    }

    public void setUsername(String username) {
        this.username = username;
    }
}
```

Figure 1.2: Example of a POJO.

examples of implementations utilizing this interface include Hibernate (post version 3.0), OpenJPA and EclipseLink.

1.1.5 ORM Software

ORM or object-relational mapping software is used to convert complex data types such as Java objects into data that is more easily stored such as database data. ORMs typically work by mapping relational database tables to object attributes and database relations into appropriate objects depending on the type of the relation [6].

1.2 Previous research

Persistence technologies of various natures come with their own advantages and disadvantages.. While such persistence layers often ease development by decreasing the amount of coded needed, they can also result in poor database design and excessive overhead when their features are used incorrectly. Various comparisons of current persistence technologies can be found as published works and online. Examples of this include a comparison of Hibernate ORM and Versant ODBMS by Mikael Kopteff [7] where features and performance are compared; and the JPA Performance Benchmark [8] which contains comprehensive benchmark results across a variety of JPA implementations with various DBMS systems.

The methods of comparison used by the aforementioned works typically involve developing implementations of select persistence solutions and running a selection of commonly used queries. The resulting execution times offer insight into the running efficiency of each solution and can be used as a benchmark. Features are

also compared through a study of what the persistence solutions offer in terms of functionality.

Previous research on persistence technologies have not covered solutions of different paradigms however. The aforementioned examples only compare ORM frameworks and frameworks which follow the JPA standard.

1.3 Problem formulation

This paper will investigate functionality and performance of various database persistence frameworks currently found in the Java world. Different persistence technologies focus on enhancing different aspects of software development such as implementation efficiency, extensibility and maintainability. In doing so they often incur certain overhead and trade-off performance for simpler development. Due to this, finding the most suitable solution will always vary based on the requirements and circumstances of a project. The following persistence solutions will be compared:

- Hibernate 5.1.0
- OpenJPA 2.4.1
- MyBatis 3.3.1
- JDBC

These candidates were chosen to cover different persistence paradigms such as ORMs and more manual solutions (which in this case is MyBatis) alongside a JDBC implementation as a point of reference.

While JDBC itself cannot be considered a persistence framework as it is an API for accessing relational databases using Java, it may be referred to as such in the following sections for the purpose of more uniform discussion.

1.4 Motivation

With the advent of object oriented programming languages and their emphasis on usage of class based objects which are typically non-scalar in nature, storing this type of data has become more complex. For example, consider a product listing on a web based commerce platform. An object oriented representation of a product may include its name, an array or dictionary of retailers with the prices offered by each individual retailer, stock status and more. Whereas the ”Product” is treated as a single object in a programming language, most modern databases with the exception of certain NoSQL solutions would not be able to store a ”Product” as a single object. Instead it must be converted into a set of simple values and ideally, have its complex fields such as the retailers split off into separate tables and referenced with a relation instead. Persistence layer software can help bridge this gap between the programming implementation of the object with its storage format.

This paper will serve to compare various existing persistence technologies to provide an overview on their advantages and disadvantages; thus giving an understanding on which persistence solutions are better suited to which types of situations.

1.5 Research Question

The research questions I will be attempting to answer are the following.

RQ1	How is SQL query generation handled in the candidate persistence frameworks?
RQ2	How do the candidate persistence frameworks compare in terms of implementation complexity?
RQ3	To what extent does performance differ between the candidate persistence frameworks?
RQ4	What potential benefits do the candidate persistence frameworks offer in software development?

I expect the frameworks which increases abstraction such as ORMs to decrease implementation complexity by reducing the amount of code needed. They should however also receive more impact to their performance due to greater overhead in executing queries. While features such as automated mapping between Java classes and database tables help alleviate complex implementations, they also hinder developers from having a great degree of direct control on the source database or how it is utilized in code.

1.6 Scope/Limitation

For this paper focus will be placed on the performance and implementation complexity of four persistence frameworks; for this purpose I will not be delving into comparing the numerous sets of functions and features that may be offered. While the number of persistence frameworks available for the Java language far outnumber those that have been selected, implementation is time consuming and having a smaller candidate pool will allow for a more concise comparison. Additionally any form of external feedback based such as surveys and interviews will be out of scope for the comparison in implementation complexity as it will require multiple test individuals to participate in a study; which is too large a project to be handled given the time frame and resources of this project.

1.7 Target group

This research can be of help to developers looking to use persistence layer software in projects. It provides an overview of various persistence solutions and a comparison of their performance, detailing how difference persistence frameworks make tradeoffs between performance and ease of use. It will also give readers an understanding of which types of persistence solutions are suitable for which situations.

1.8 Outline

The method chapter will outline how the comparison will be carried out and discuss any potential issues they may cause. The implementations chapter will provide an overview of the implementation process and some code snippets to explain how the frameworks are used. This will be followed up by a result chapter and analysis which will provide the data obtained in our experiments and an analysis of the data respectively. A discussion chapter will serve to discuss our findings from the perspective of our research questions. Finally a conclusion chapter will provide a

wrap up to this paper and also make notes of possible improvements and future research.

2 Method

This chapter will describe the scientific approaches and methods that will be used in attempting to answer the research questions.

2.1 Scientific approach

As the research questions presented in this paper are multifaceted in the qualities they measure I will be conducting both inductive and deductive research to attempt to answer them. Research question 1 will be answered through theoretical research as we can deduce its answer from various documentation for the persistence frameworks. For research questions 2 and 3 that require quantitative data I will be creating implementations of the persistence frameworks through which I shall conduct experiments and present the results. The final question will be answered in a reflective manner whereby experiences gained in the preparation and execution of the experiments will be used as a benchmark.

2.2 Method description

Initial research will consist of reading documentation and exploring the functional aspects of the persistence frameworks. Sources for the documentation will be restricted to official documentation to ensure reliable and accurate information. An implementation of each persistence technology will be then be created for both familiarizing with their functionality as well as for preparation of an experiment to measure their performance. The implementations will all be done with the same target database schema shown in Figure 2.1. The schema is designed as such that it is simple to implement and easy to understand, yet still contains design patterns such as one-to-many and many-to-many relationships so that they may be used in testing the persistence implementations.

To ensure a fair comparison of implementation complexity, each implementation will need to be able to achieve the following in functional requirements:

- Retrieve one or more rows of tables `user`, `post`, `comment` and `tag` and return them as a (list of) the corresponding Java class instance.
- Filtering results for the aforementioned tables.
    1. Filter all tables except `post_has_tags` by `id`.
    2. Filter `post` by `user_id` and related `tags` by `id` or `name`.
    3. Filter `comment` by `user_id` and `post_id`.
    4. Filter `tag` by `name`.
    5. Filter `user` by `username`.
- Foreign keys columns within database shall be replaced with an object instance of the corresponding foreign key relation and must be resolved eagerly.

  i.e. `post` table's `user_id` column shall be replaced with a `User` Java object in a `Post` Java object and it must be retrieved together with a `Post`.

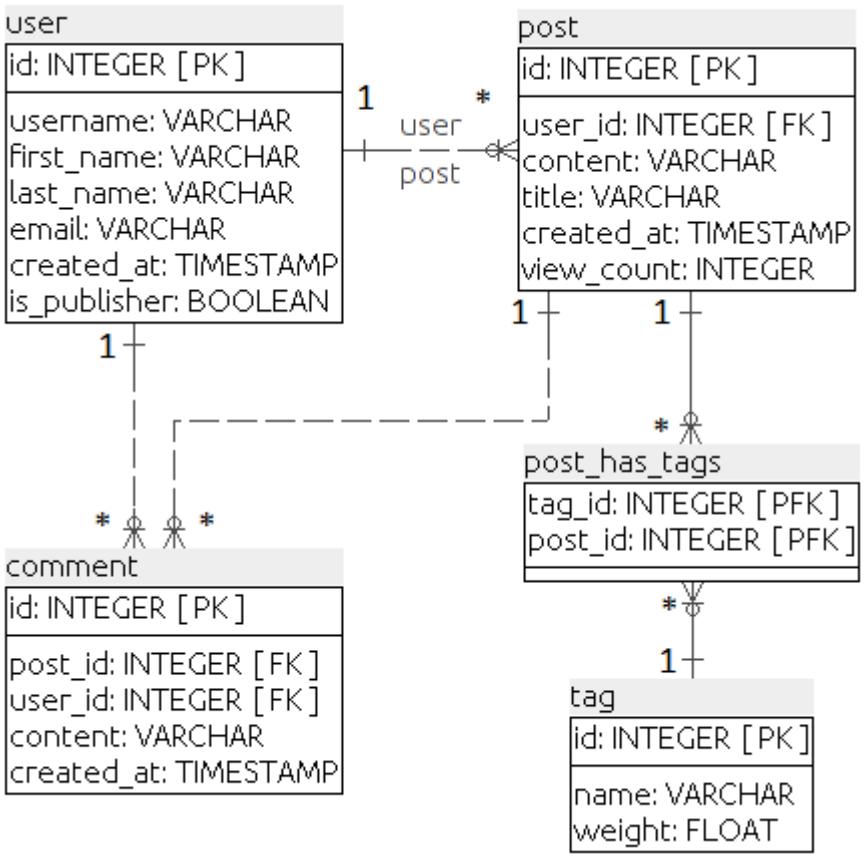

Figure 2.1: Implementation database schema.

- Many-to-many relations shall be represented using a Java collection and must be resolved eagerly.

  i.e. `post_has_tags` shall be represented by a collection of `Tag` objects within the `Post` Java object and they must be retrieved together with a `Post`.

- Insert and delete one row of tables `user`, `post`, `comment` and `tag`.

- Update one row of tables `user`, `post`, `comment` and `tag` with any combination of editable columns.

An experiment will be conducted by running a predetermined set of queries through each of the persistence implementations; the results that will be extracted and analysed from this experiment will be the time taken for the query to be executed and the query results returned. Each query will be run multiple times and an average be used as the result.

The experiment queries will be as follows:

- SELECT query with large result set (5001 rows)
- SELECT query with small result set (6 rows)
- SELECT query with single result
- SELECT query filtering on foreign key

- SELECT query utilizing a single join
- SELECT query utilizing two joins
- INSERT single row
- UPDATE single row (single column update)
- DELETE single row (no cascade)

Each query will be run using a test runners that will follow the algorithm shown in Figure 2.2. This will be done to isolate the test query from external disturbances and ensure that no data is actually changed in the database between test runs.

```
foreach test:
    if test is select:
        start = currentTime;
        run test;
        result = currentTime - start;
        clear cache;
    else:
        begin transaction;
        start = currentTime;
        run test;
        result = currentTime - start;
        roll-back transaction;
        clear cache;
```

Figure 2.2: Experiment algorithm

The implementation complexity of each framework will be measured by counting the total lines of code written per implementation; this includes source code and configuration files.

2.3 Reliability and Validity

To attempt to increase reliability of the performance comparing experiment I will set up the experiment environment so that it suffers minimal external disturbances. To achieve this the experiments will be conducted within a virtual machine running a clean installation of the Ubuntu operating system. Each persistence implementation will be tested with the same target database and transactions will be utilized to ensure that the data will not be affected in any way between experiments.

2.4 Ethical Considerations

As this paper is focused on implementation and testing of software frameworks there are no foreseeable relevant ethical considerations.

3 Implementations

This section will provide an overview of the individual software candidates and describe the implementations. All frameworks are implemented towards a common database schema shown in Figure 2.1. All implementation and following experimentation are carried out on a virtual machine running Ubuntu 14.04 64-bit on a Windows 10 64-bit host. IntelliJ IDEA 15.0.4 Community Edition is used as the project IDE for development and compilation and PostgreSQL 9.3.11 is used as the database backend.

3.1 Hibernate

Hibernate is an ORM framework originally started in 2001. Since its version 3.0 release it has become JPA certified for API 2.0 in addition to providing its own native API. Hibernate utilizes HQL or Hibernate Query Language in place of SQL to provide querying capabilities against its own data objects in a high level manner rather than interact directly with the database. Hibernate provides support for performance tuning through features such as lazy initialization/loading and various fetching strategies for handling relationships [9].

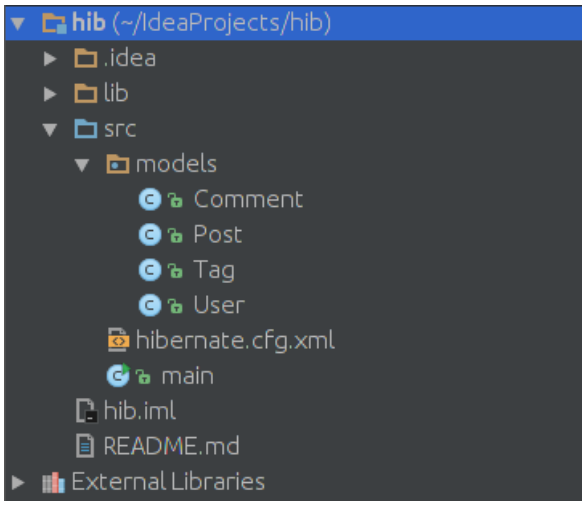

Figure 3.1: Hibernate implementation project structure.

3.1.1 Implementation

The Hibernate implementation consists of a configuration file and object classes. The configuration file hibernate.cfg.xml (shown in Figure 3.2) holds the information pertaining to target database type, connection details, class mappings and Hibernate specific settings. The classes given in the mapping are POJOs marked with annotations to define persistence properties. The file structure for the project is shown in Figure 3.1.

The `@Entity` annotation is used to designate the class as a persistence entity as shown in Figure 3.3. It is followed by the `@Table` annotation which sets the database table it is to be persisted; this may be omitted if the class name matches the table name.

`@Id` declares our integer `id` as the identifier property of the post entity. We also denote that this field's value is auto-generated by annotating it with `@GeneratedValue`.

```
<hibernate-configuration>
    <session-factory>
        <property name="hibernate.dialect">
            org.hibernate.dialect.PostgreSQL9Dialect
        </property>
        <property ...
            ...
        />
        <mapping class="models.Comment"/>
        <mapping class="models.Tag"/>
        <mapping class="models.User"/>
        <mapping class="models.Post"/>
    </session-factory>
</hibernate-configuration>
```

Figure 3.2: Hibernate configuration file snippet.

```
@Entity
@Table(name="post")
public class Post {
    ...
}
```

Figure 3.3: Defining an entity and mapping to a table.

```
@Id @GeneratedValue(strategy=GenerationType.IDENTITY)
@Column(nam ="id")
private int id;
```

Figure 3.4: Defining an identity property, auto-generation and mapping to a column.

Finally, `@Column` denotes the name of the corresponding column in the database table. Figure 3.4 shows how these annotations were used to for and id column.

```
@Column(name="created_at",insertable=false,updatable=false)
private Timestamp created_at;
```

Figure 3.5: Defining column-specific constraints.

Constraints can also be defined in the `@Column` annotation. The `created_at` field in the `post` model is a generated timestamp which is automatically handled by the database. Disallowing insertion and updating of this field from hibernate can be done by passing parameters as shown in Figure 3.5.

As shown in Figure 3.6, foreign keys can be referenced using the appropriate annotations. `@OneToOne` relations are defined by providing the foreign key column and declaring the field as the related Java class. In this case a `User` object is referenced through the foreign key field `user_id`. When the object is fetched from

```
@OneToOne
@JoinColumn(name="user_id")
private User user;

@ManyToMany(targetEntity=Tag.class, fetch=FetchType.EAGER)
@JoinTable(
        name="post_has_tags",
        joinColumns=@JoinColumn(name="post_id"),
        inverseJoinColumns=@JoinColumn(name="tag_id")
)
private Collection tags;
```

Figure 3.6: Defining one-to-one and many-to-many relationships.

the database Hibernate will automatically resolve the `User` object either immediately or lazily depending on configuration.

`@ManyToMany` relations are more complex than one-to-one relations. The intermediate table holding the primary keys of both models is defined in the `@JoinTable` annotation along with the foreign key columns. To achieve eager resolution of a many-to-many relation the parameter `fetch` is set to `FetchType.EAGER`. Doing so ensures that Hibernate will fetch the related objects prior to returning the query results. This was not done in our case with one-to-one relations as they are eagerly resolved by default.

3.1.2 Usage

Hibernate enables data interaction through `Session` objects which are in turn created by a `SessionFactory` generated with the help of the configuration file. `Session` objects provide multiple interfaces for retrieving, persisting, updating and deleting objects. Figure 3.7 shows how a `Session` is used in retrieving a single or a list of `User` object(s).

```
Session session = sessionFactory.openSession();

// Using HQL to get objects.
List<User> users = session.createQuery("from User where id
   in (1,2,3)").list();

// Using load to retrieve user with id 1.
User user1 = (User) session.load(User.class, 1);

session.close()
```

Figure 3.7: Using `Session` objects to retrieve data.

A new entity may be persisted to database by initializing a Java object and using the `persist()` or `save()` method as shown in Figure 3.8; though it is advised to use the former due to it functioning in accordance with transactions, the latter will execute an insert statement whether it is within our outside a transaction and can

cause issues in data consistency. One difference to take into consideration is that `persist()` is a void method whereas `save()` will return the generated identifier of the instance which may be useful in certain scenarios.

```
// Using persist()
User user = new User("Andy", ...);
session.persist(user);

// Using save()
Post post = new Post(user, "A new post", ...);
int id = session.save(post);
```

Figure 3.8: Persisting a new entities.

Updating an object can be done in two ways depending on the its state as shown in Figure 3.9. An object is considered persistent if it is linked to a `Session` object and detached if not. A persistent object can be manipulated by modifying its fields in its Java representation; these changes are automatically pushed to the database when the `Session` is flushed. A detached object be updated by reattaching the object using `merge()` and then updated, or the object can be updated directly using HQL.

```
// Updating a persistent object.
User user = (User) session.load(User.class, 1);
user.setUsername("new_username");
session.flush();

// Updating using HQL.
session.createQuery("update User u set u.username=\"
   new_username\" where u.id=2").executeUpdate();
```

Figure 3.9: Updating entities.

Deleting objects can be done through the API method `delete()` or by using HQL as shown in Figure 3.10. Batch deletion can only be performed using HQL.

```
// Deleting using delete().
Comment comment = (Comment) session.load(Comment.class, 1);
session.delete(comment);

// Deleting using HQL.
session.createQuery("delete Comment c where c.user_id=2").
   executeUpdate();
```

Figure 3.10: Updating entities.

3.2 OpenJPA

The Apache OpenJPA ORM is an open source implementation of the JPA specification. It is similar in usage to Hibernate with which it shares many features such as using annotations to denote persistence properties and layout of the configuration file. One major difference between them is that OpenJPA utilizes JPQL rather than HQL as its querying language [10].

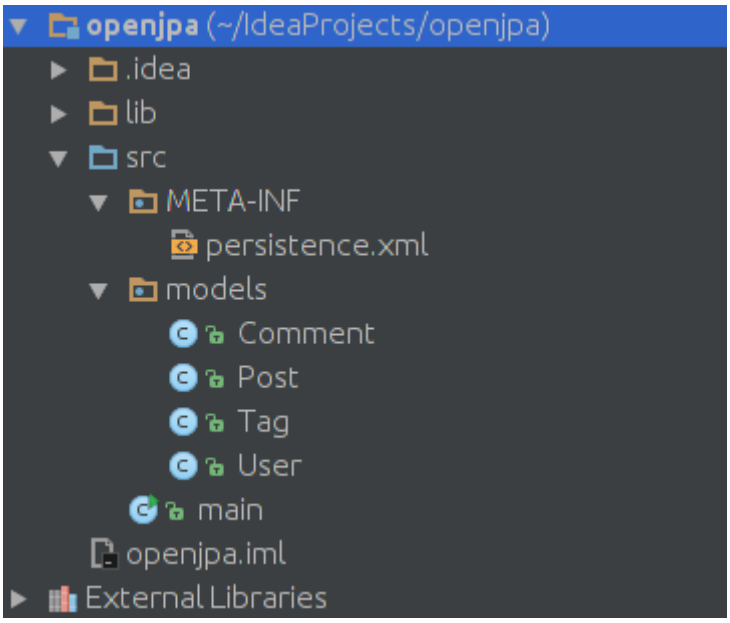

Figure 3.11: OpenJPA implementation project structure.

3.2.1 Implementation

Similar to the Hibernate implementation, the OpenJPA implementation makes use of a configuration file and model class files. The configuration file is named persistence.xml and contains data required to connect to the database and the Java classes to be used. The implementation structure is also very similar with the exception of the configuration file placement as shown in Figure 3.11.

One instance of differing usage from Hibernate was encountered when editing the configuration file. Unlike Hibernate where properties and model mappings could be defined in any order, OpenJPA configuration files must follow a strict order of elements as shown in Figure 3.12. Class mappings must be defined prior to any properties or else they will not be found when the configuration is read. OpenJPA also supports multiple `persistence units` which can be considered persistence environments.

The remaining implementation of OpenJPA is near identical to the Hibernate implementation and thus will not be discussed further.

3.2.2 Usage

While a majority of functions are performed in similar fashion in OpenJPA through the JPA API like Hibernate, there are a few differences in usage. Rather than using `Session` objects, OpenJPA utilizes a `EntityManager` object shown in Figure 3.13 to manage data. `EntityManager` must be instantiated with a persistence unit defined

```
<persistence-unit name="openjpa">
    <provider>org.apache.openjpa.persistence.
       PersistenceProviderImpl</provider>
    <class>models.Comment</class>
    <class>models.Post</class>
    <class>models.Tag</class>
    <class>models.User</class>
    <properties>
        <property name="openjpa.ConnectionDriverName" value=
           "org.postgresql.Driver"/>
        ...
    </properties>
</persistence-unit>
```

Figure 3.12: OpenJPA configuration file snippet.

in the configuration file and provides a similar API to the Hibernate `Session`. Some of the differences in usage are shown in the Figure.

```
// Creating a new EntityManager.
EntityManager em = Persistence().createEntityManagerFactory(
   "openjpa").createEntityManager();

// Retrieve single object using find().
User user = em.find(User.class, 1);

// Retrieving objects using JPQL.
List<User> users = em.createQuery("SELECT u FROM User u
   WHERE u.id in (1,2,3)").getResultList();

// Deleting an object using remove().
em.remove(user);

em.close();
```

Figure 3.13: Instantiating and utilizing an `EntityManager`.

3.3 MyBatis

MyBatis is a persistence framework forked from another persistence framework iBATIS 3.0. It differs from the previous ORM persistence frameworks in that it does not map Java classes to database relations but rather maps user defined queries and statements to Java methods [11]. This means that all SQL queries must be written manually, making it more time consuming to implement than ORMs but also providing more control to developers in regards to exactly how data is accessed and manipulated. This can be seen in the project structure overview in Figure 3.14 which contains far more files than the previous ORM implementations.

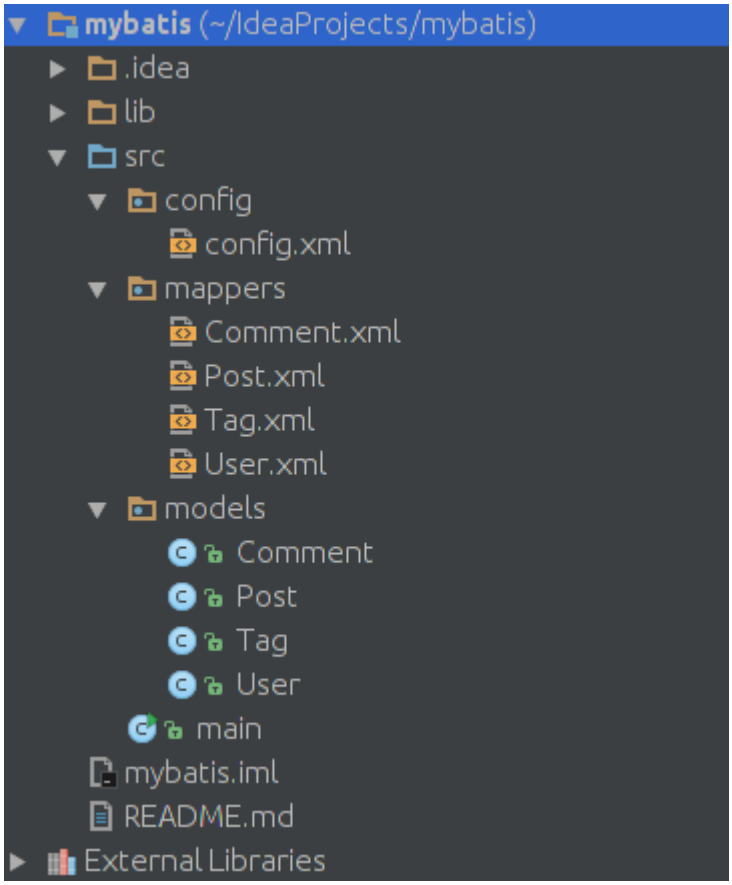

Figure 3.14: MyBatis implementation project structure.

3.3.1 Implementation

MyBatis makes use of mappers which can either be implemented as interfaces or standalone xml files to determine how persistence is achieved. As SQL statements can become very long in complex queries it was decided to separate the mappers into xml files for easier management. The mapper xml files are divided according to the object class that they represent or effect. For instance, the `Post.xml` mapper contains the `resultMap` and queries that act on the `post` table.

`ResultMaps` define the mapping between query results and its Java POJO class and is one of the most powerful elements in MyBatis. Each `id` and `result` element maps a query result column to a field within the POJO. The `id` element designates the column as the identifier column in the database relation and helps increase overall performance. The *association* elements maps a foreign key column to a nested

```
<resultMap id="PostResult" type="Post">
    <id property="id" column="id"/>
    <result property="content" column="content"/>
    <result property="title" column="title"/>
    <result property="created_at" column="created_at"/>
    <result property="view_count" column="view_count"/>
    <association property="user" resultMap="User.
       NestedUserResult"/>
    <collection property="tags" javaType="ArrayList" ofType=
       "Tag" column="id" select="Tag.getTagsByPost"/>
</resultMap>
```

Figure 3.15: Example of a `ResultMap`.

`resultMap` which in the case of Figure 3.15 is a `User`. Finally the `collection` element maps a collection of many-to-many relations by referencing the related object class and a query.

```
<select id="getPostById" parameterType="int" resultMap="
   PostResult">
    SELECT
        ...
    FROM public.post p
        INNER JOIN public.user u
            ON p.user_id = u.id
    WHERE p.id = #{id};
</select>
```

Figure 3.16: Defining a select statement with a single integer parameter.

Each of the `select`, `insert`, `update` and `delete` elements maps a statement of a corresponding type. The `select` statement shown in Figure 3.16 for instance retrieves data with an integer parameter that is used for filtering; this is referenced in the actual query by `#{variable_name}` within the string. Each resulting data rows from the query is then used to populate a `resultMap`, which in turn is a mapped to the `Post` Java class. Mapped queries are invoked by referencing mapped object type followed by the `id` of the statement. The statement shown in Figure 3.16 is referenced by the string `Post.getPostById` and will return a list of `Post` objects.

Statements can be dynamically generated using additional XML elements. Figure 3.17 shows usage of a loop which can help build an array in the query. The parameter taken by this query is a Java array; the loop will generate an array in which each entry is separated by a comma and the entire array surrounded by parenthesis.

Conditionals in mapped queries can be helpful in situations when it is not known if a certain parameter will be provided. Figure 3.18 shows how this is useful when mapping an `update` statement as multiple fields can be updated at the same time. The conditional element helps avoid having to write multiple `update` statements based on which columns are being changed.

```
<select id="getPostByIds" resultMap="PostResult">
    SELECT
        ....
    FROM public.post p
    WHERE p.id in
    <foreach item="id" index="index" collection="array" open
       ="(" separator="," close=")">
        #{id}
    </foreach>;
</select>
```

Figure 3.17: Dynamically generated block in select statement.

```
<update id="updatePost">
    UPDATE public.post
    SET
    <if test="title!=null">
        title = #{title},
    </if>
    <if test="content!=null">
        content = #{content},
    </if>
    <if test="view_count!=null">
        view_count = #{view_count}
    </if>
    WHERE id = #{id};
</update>
```

Figure 3.18: Dynamically generated block using conditionals.

3.3.2 Usage

Once the persistence classes and mapper elements have been defined, MyBatis provides a Java API through `SqlSession` objects for interacting with data. `SqlSession` objects can be generated using an `SqlSessionFactory` in similar fashion to Hibernate. The process for obtaining a session can be seen in Figure 3.19.

```
// Opening session.
SqlSession session = sessionFactory.openSession();
...
session.commit();
session.close();
```

Figure 3.19: Generating and utilizing an `SqlSession`.

Methods are provided per statement type i.e. select, insert, update and delete. Select can be done through three different methods depending on what is needed. Figure 3.20 shows how `selectOne()`, `selectList()` and `selectMap()` may be utilized.

```
// Selecting all users.
List<User> all_users = session.selectList("User.getAll");

// Selecting multiple users.
int[] user_ids = new int[] {1, 2, 3, 4};
List<User> users = session.selectList("User.getUsersByIds",
    user_ids);

// Selecting single user.
User user = session.selectOne("User.getUserById", 5);

// Selecting users into id:User map.
HashMap<Integer, User> user_map = session.selectMap("User.
    getAll", "id");
```

Figure 3.20: Different methods for retrieving data.

Figure 3.21 shows how other query types are used. The remaining methods follow a similar usage pattern, taking the query id string and an object or field as parameter depending on how the query is defined in the mapper. For instance, the delete method can be changed to take a `User` object as input and retrieve the `id` field automatically as an SQL parameter.

```
// Inserting a new user.
User user = new User("Jack", ...);
int new_user_id = session.insert("User.insertUser", user);

// Updating an existing user.
user.setUsername("John");
session.update("User.updateUser", user);

// Deleting a user.
session.delete("User.deleteUser", user.getId());
```

Figure 3.21: Different methods for retrieving data.

3.4 JDBC

The JDBC (Java Database Connectivity) API is a Java application programming interface that defines how the programming language interacts with databases. It provides an interface through which statements can be created, modified, executed and also provides support for managing database transactions.

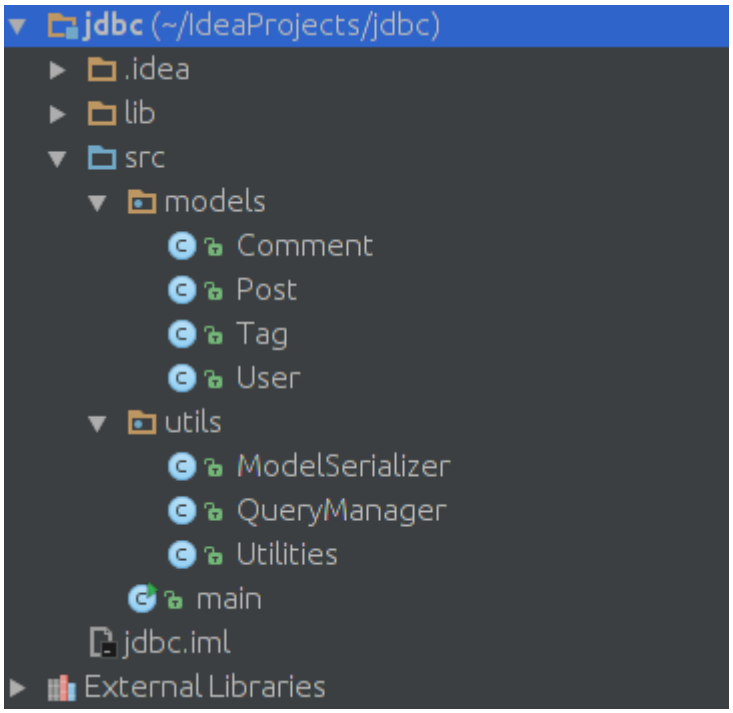

Figure 3.22: JDBC implementation project structure.

The JDBC implementation structure shown in Figure 3.22 was kept simple to minimize the amount of overhead that may be created with additional abstraction. Similar to the other implementations POJOs are used as representations of the objects held in the database.

The `ModelSerializer` class hold static methods that take a result set returned from executing a query and converts their contents into Java objects. As can be seen in Figure 3.23, this is done by looping through the the `resultset` and instantiating a new object for each row. This is then placed into a `HashMap` where the key is the integer `id` of the object and the value is the object itself. A `HashMap` is utilized so that optimizations can be carried out for fetching related objects relations

The `QueryManager` class utilizes a JDBC `Connection` provided during instantiation to make queries. `PreparedStatement` is used consistently as good practice due to faster overall performance and the many vulnerabilities of `Statement` such as SQL injection. Figure 3.24 shows how a list of `Comment` objects is retrieved by first executing a `PreparedStatement` and passing the results to the serializer method. In the case of `Comment` objects the functionality provided by `HashMap` is not utilized directly as we simply return its values as the result; Figure 3.25 shows how we utilize this when appending a `Post` object's list of `Tag` objects. Having stored the `Post` objects in a `HashMap`, we now have very quick access to a list of all id values. This allows us to retrieve `Tag` objects belonging to all `Post` objects in the result list in a single query and bypass the N+1 query issue.

```
public static HashMap<Integer, Post> parsePostList(ResultSet
    rs) {
    HashMap<Integer, Post> ret = new HashMap<>();
    try {
        while (rs.next()) {
            Post post = new Post();
            post.setId(rs.getInt("id"));
            // Set remaining fields of post object.
            ...

            User postUser = new User();
            postUser.setId(rs.getInt("user_id"));
            // Set remaining fields of post's user object.
            ...

            post.setUser(postUser);
            ret.put(post.getId(), post);
        }
        return ret;
    } ...
}
```

Figure 3.23: Serialization process for a list of `Post` objects.

```
public List<Comment> getCommentsByIds(int[] ids) {
    try {
        String query = "SELECT c.id, " ... +
                // Append additional fields for retrieving
                   comment and related objects.
                "FROM public.comment c " +
                "JOIN public.user u ON c.user_id = u.id " +
                "JOIN public.post p ON c.post_id = p.id " +
                "WHERE c.id in " + Utilities.arrayToString(
                   ids) + ";";
        PreparedStatement st = conn.prepareStatement(query);
        ResultSet rs = st.executeQuery();
        HashMap<Integer, Comment> res = ModelSerializer.
           parseCommentList(rs);
        return new ArrayList<>(res.values());
    } ...
}
```

Figure 3.24: Retrieval process of a list of `Comment` objects.

The `Utilities` class holds various static methods such as converting a Java set or array into a string representation that can be parsed by the database.

Overall the JDBC implementation required far more code to implement than the previous frameworks discussed as all SQL statements must be manually written resulting in large amounts of repeated code for the various object types. However

```
public List<Post> appendPostTags(HashMap<Integer, Post>
    posts) {
     if (posts.size() == 0) {
         return new ArrayList<>();
     }

     try {
         Set<Integer> post_ids = posts.keySet();
         String query = "SELECT pt.post_id, " ... +
                 "FROM public.post_has_tags pt " +
                 "JOIN public.tag t ON pt.tag_id = t.id " +
                 "WHERE pt.post_id in " + Utilities.
                    setToString(post_ids) + ";";
         PreparedStatement st = conn.prepareStatement(query);
         ResultSet rs = st.executeQuery();
         HashMap<Integer, Post> res = ModelSerializer.
            appendPostTags(posts, rs);
         return new ArrayList<>(res.values());
     } ...
}
```

Figure 3.25: Optimizations for resolving a many-to-many relation.

this provides an opportunity to optimize functions and queries for better control of the work flow.

4 Results

This chapter will provide the aggregated results obtained during the experiments as well as the final code length of each implementation.

Query	Hibernate	OpenJPA	MyBatis	JDBC
Large select	152.16	109.35	131.41	63.78
Small SELECT	2.3	1.72	2.76	1.38
Single SELECT	0.16	0.15	0.28	0.36
Single JOIN	60.57	12.13	71.54	8.23
Double JOIN	62.07	10.77	72.6	10.33
Insert	0.31	0.2	0.22	0.28
Update	0.16	0.5	0.47	0.41
Delete	0.17	0.37	0.16	0.23

Table 4.1: Execution times in milliseconds rounded to the second decimal.

Each query was run a total of 50 times with every persistence implementation. The resulting time for each was taken by averaging the last 45 results. This was done to mitigate the effects of PostgreSQL's own caching capabilities which greatly reduced response times when a query is run more than once. The results have been divided into separate graphs for easier viewing. Figure 4.1 aggregates all results of select queries and Figure 4.2 aggregates the remaining queries.

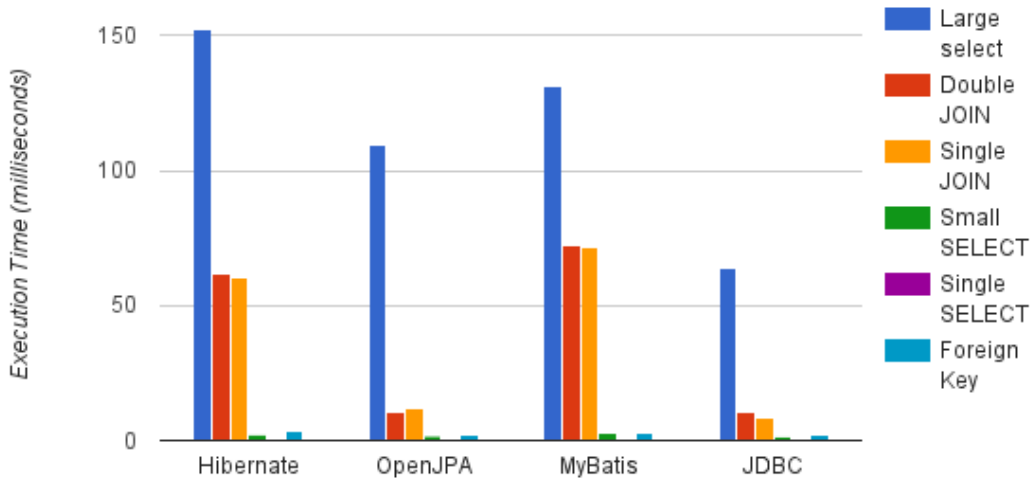

Figure 4.1: Execution times for SELECT statements.

Figure 4.1 shows largely varying SELECT statement execution times between the implementations with JDBC having generally fastest times followed by OpenJPA. Both MyBatis and Hibernate implementations are visibly slower with certain

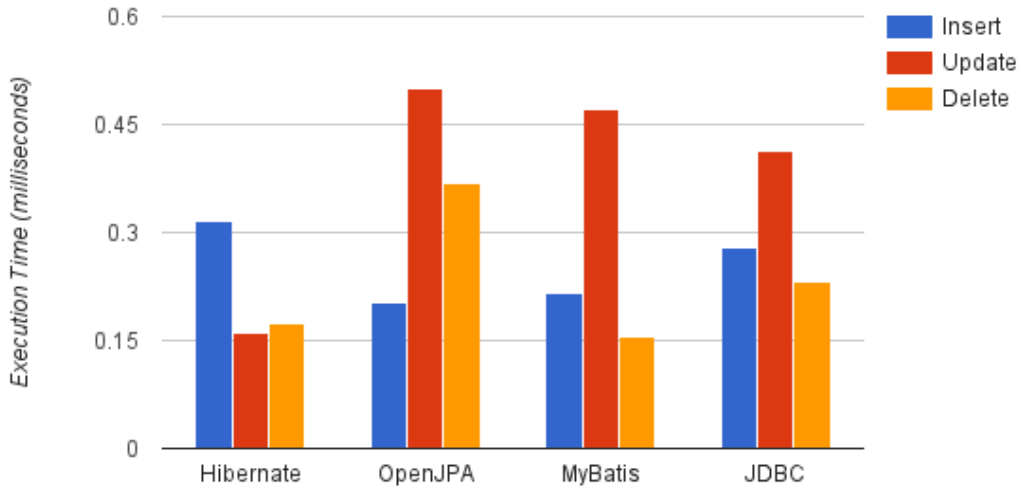

Figure 4.2: Execution times for INSERT, UPDATE and DELETE statements.

statements e.g. single and double join taking over three times longer to execute compared to JDBC.

Figure 4.2 shows the OpenJPA, MyBatis and JDBC implementations to all have relatively similar performance in UPDATE and INSERT statements. The Hibernate implementation had the fastest execution times for UPDATE but the slowest in INSERT. Execution times for DELETE statements showed MyBatis and Hibernate to be the quickest followed by JDBC and OpenJPA.

The number of lines of each code shown in table 4.2 include source code and configuration for the implementations. Running code for the experiments are not included as they do not constitute part of implementation.

Code	Hibernate	OpenJPA	MyBatis	JDBC
Source Code	323	325	807	1116
Configuration	35	21	30	0
Total	358	346	837	1116

Table 4.2: Lines of code for each implementation.

5 Analysis

This section will provided an analysis of the results obtained the performance experiment. Each query will be discussed separately to provide more in-depth information.

5.1 Large SELECT query

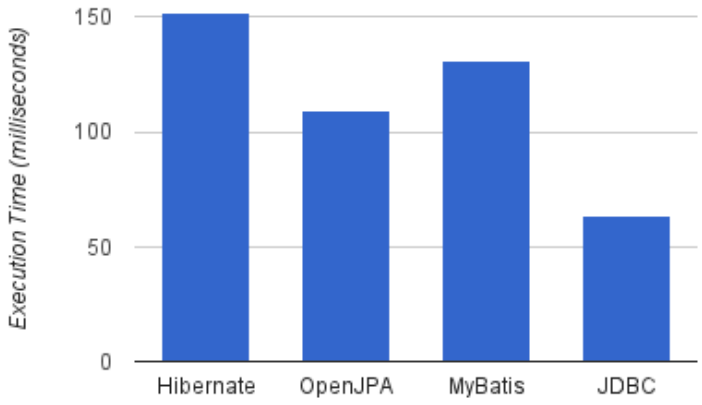

Figure 5.1: Results for large select query.

This query retrieves all rows from the `comment` table totalling 5001 entries. Hibernate performed slowest taking over 152 ms to return complete execution. MyBatis and OpenJPA follow with 131 ms and 109 ms respectively and JDBC finishing quickest at 63 ms.

5.2 Small SELECT query

The small select query retrieves six rows from the `post` table by filtering with predetermined `id` values. Here we see in Figure 5.2 that the MyBatis and Hibernate implementations perform significantly worse than the other candidates. Logging the queries during runtime revealed that the both implementations suffered from the N+1 query optimization issue as it made an additional query to retrieve the tags for every `Post` result. Interestingly despite having set the many-to-many relation of `Post` object's related `Tag` list to be resolved eagerly in OpenJPA and Hibernate, OpenJPA does not load them directly and maintains a lazy fetch type.

5.3 Single SELECT

The single select query retrieves one row from the `user` table by an `id` value. Figure 5.3 shows that the ORM implementations perform the quickest; possibly due to optimizations in single row retrieval. The JDBC implementation's poor performance may be attributed to it having to parse the result as a list regardless of the number of query results; this may be fixed by adding methods that specifically return a single instance for a query.

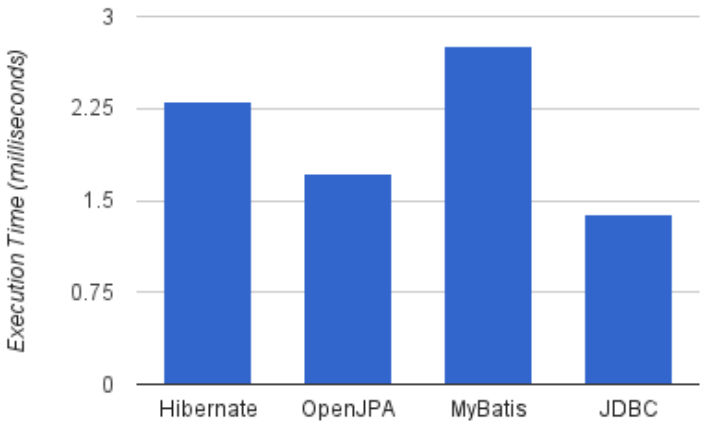

Figure 5.2: Results for small select query.

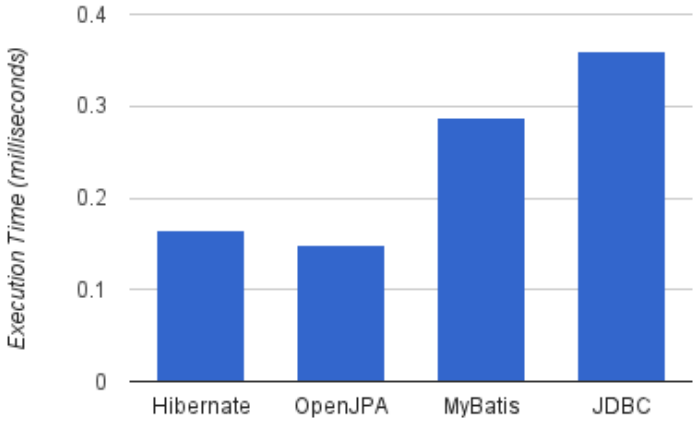

Figure 5.3: Results for single select query.

5.4 Single JOIN SELECT

The single join select query retrieves rows from the `post` table by joining with `post_has_tags` and finding entries with predetermined `tag_id` values. Similar to the small select query, MyBatis and Hibernate once again suffers a massive performance penalty due to the N+1 query issue as seen in Figure 5.4. Additionally, the OpenJPA implementation once again fails to eagerly resolve the `Post`/`Tag` relation thus resulting in very fast execution time. The JDBC implementation comes in fastest due to manual optimizations set in place during development where all `Tag` objects for a list of `Post` objects are retrieved with a single query.

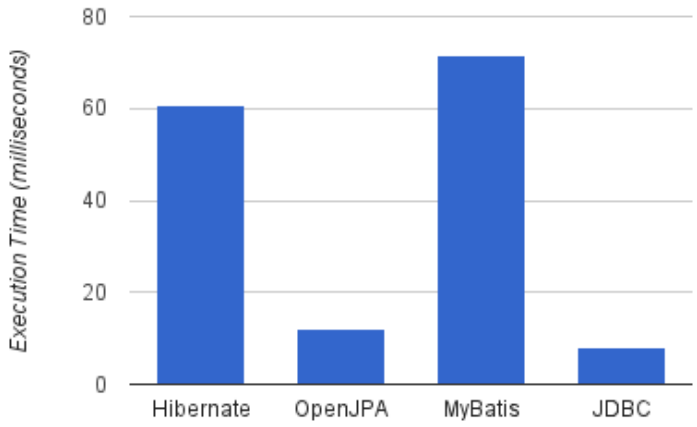

Figure 5.4: Results for select query with one join.

5.5 Double JOIN SELECT

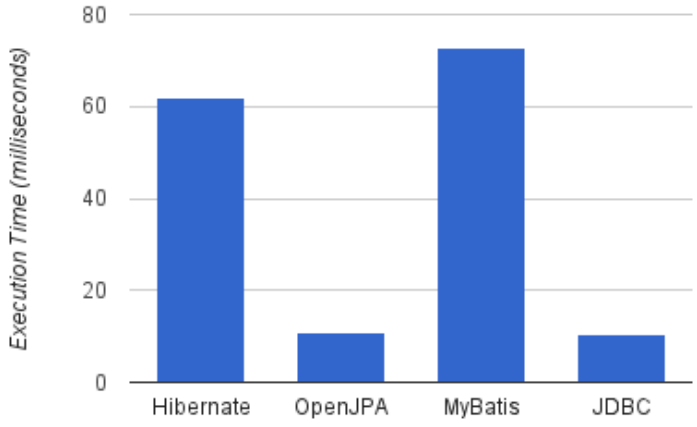

Figure 5.5: Results for select query with two joins.

The double join query extends upon the single join query by creating a three-way join between the tables `post`, `post_has_tags` and `tag` and finds `post` entries with predetermined `tag` names. The trend of long execution times due to N+1 query issue continues here as we see both Hibernate and MyBatis being significantly slower than the OpenJPA and JDBC implementations in Figure 5.5.

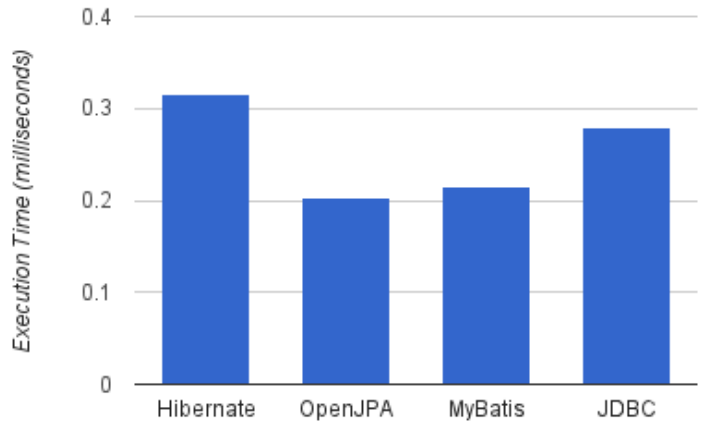

Figure 5.6: Results for single row insert.

5.6 INSERT

The insertion statement adds a new row into the `user` table. Departing from the large disparities of the previous experiments, most implementations performed comparably as shown in Figure 5.6. An oddity is that despite both being ORMs, the OpenJPA implementation executes roughly 30% faster than Hibernate implementation.

5.7 UPDATE

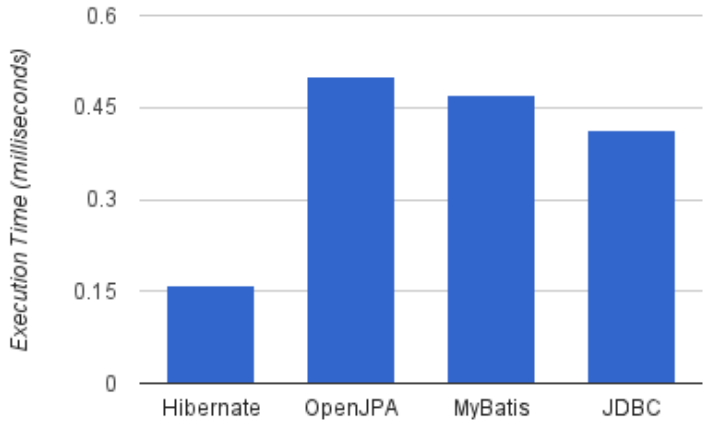

Figure 5.7: Results for single row update.

The update statement updates the `username` column of a single row in the `user`

table. Figure 5.7 shows that in this experiment the Hibernate implementation executes almost two and a half times faster than the JDBC implementation which is second fastest. Logging queries during the experiment revealed that the Hibernate implementation had not actually issued any SQL to the database and thus was returning without having completed the update action prior to rolling back the statement. Research into this issue revealed that Hibernate has optimizations whereby updates are not carried out until a transaction is committed; this caused issues as transactions in the experiment must always be rolled back to preserve state of the data for repeated runs. The other implementations performed comparably.

5.8 DELETE

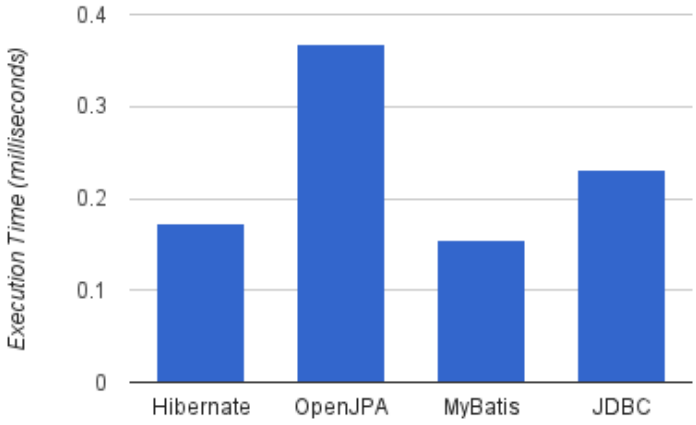

Figure 5.8: Results for single row delete.

The delete statement removes a single row from the `comment` table. Figure 5.8 shows the Hibernate and MyBatis implementations performing similarly, with JDBC and OpenJPA implementations being slowest.

5.9 Implementation Complexity

Referring back to table 4.2, we see that both ORM implementations had significantly fewer lines of code compared to the other implementations. This can largely be attributed to their efficient integration of defining persistence properties within POJOs using annotations and the automated SQL generation they provide. The MyBatis implementation contained considerably more code due to the need to manually write SQL queries. The JDBC implementation had by far the largest code base; this was due to the fact that all functionalities had to be manually implemented from the execution of queries to the parsing of query results as Java objects.

Both ORM implementations provide a majority of features needed out-of-the-box and can easily be configured should additional features be required e.g. lazy-loading for relations and options for cascading. The MyBatis implementation is also easily configurable through its mapper files although this will require more work to make

alterations to data structures and behaviour; its `resultMap` elements and dynamic query generation capabilities are powerful tools that provide reconfigurability without its operation becoming overly abstracted or becoming a black-box. The JDBC implementation is the most tedious and difficult to maintain and expand upon. Adding a new object type for example would require a host of new methods within both the `ModelSerializer` and `QueryManager` class for parsing and querying the new object. This does however have the benefit of allowing control over every action taken and far better optimized execution (time-wise) compared to the other frameworks.

6 Discussion

Through studying and implementing the various persistence technologies that were chosen, I have been able to generate and analyse both quantitative and qualitative data that have helped in forming answers to my stated research questions.

The frameworks Hibernate and OpenJPA are both ORM frameworks that help ease development by automating a large part of the persistence process. From mapping data stored within tables to automatically generating SQL for retrieving, creating, updating and deleting database entries. Through this they are able to massively decrease the amount of code that must be written to have a functional implementation, and easy code extensibility and maintainability for future development. This however is somewhat offset by having generally slower execution time due to the extra layers of abstraction; as well as poorly optimized queries at times due to the need for these frameworks to be generic and cater to a wide variety of functions and scenarios. One example of this encountered during the implementation and experimentation stage of this project is the N+1 query issue suffered by Hibernate which significantly impacted its performance.

Implementing a persistence layer from scratch using JDBC showed both the strengths and shortcomings of developing persistence software using raw SQL. The need for all functionality from writing all queries manually to parsing results and handling relations meant that the code base required to have a functional implementation was far more than if an existing framework were to be used. Code maintainability and extensibility were equally poor due to the massive amount of repeated code or similar code caused by the need to accommodate different types of objects. Examples of this can be seen in the `QueryManager` class that included an insert, update, delete and numerous select methods for the handful of tables in our test schema. Performance-wise however, the JDBC implementation consistently performed well through the experiments due to every query being controlled during development. For instance, only the JDBC implementation was able to bypass the N+1 query issue when eagerly resolving many-to-many relations (OpenJPA had also avoided this but only by failing to resolve the relation), this resulted in execution times far quicker than any of the other candidates.

MyBatis lands between the ORM frameworks and pure JDBC by providing automated mapping features not for objects but for SQL statements. While this means that implementation is more time consuming compared to the likes of Hibernate and OpenJPA, it gives more control to developers in regards to how data is accessed. For instance, the N+1 query issue encountered during testing with MyBatis can be avoided with further development by not directly defining a `collection` element within the `post resultMap`. The `Post` objects can first be retrieved using the `selectMap()` method through which we obtain a set of `id` values that can be used to retrieve related `Tag` objects in similar fashion to the JDBC implementation.

A factor that must be taken into account in evaluating the persistence solutions in this paper is the simplistic nature of the test schema. In real-world scenarios where persistence solutions are required for highly complex schema designs with large numbers of tables, relations and data structures it may not always be feasible to create an implementation using pure JDBC. Likewise, in situations where data must be processed within time constraints it may not be feasible to use highly abstracted solutions such as ORMs due to their larger overhead.

7 Conclusion

The results obtained throughout the course of implementing and conducting experiments utilizing the various persistence technologies stand in line with what had been initially hypothesised. The more complex and abstract ORM frameworks provided easy implementation and access to a variety of functionality with only a small amount of code. This contrasted the more on-hands approach taken by MyBatis which requires SQL queries to be defined manually but through which it was possible to control the querying process down every action. Performance was also impacted by the level of abstraction of each persistence implementation. With the exception of a few experiments that did not go as planned due to implementation issues the less abstracted solutions generally performed better; with the pure JDBC implementation consistently having relatively fast execution.

The N+1 query issue encountered could have been mitigated in MyBatis by redesigning retrieval process for many-to-many relations, this however is not applicable to Hibernate as its actions are not directly controllable and would require more complex workarounds.

When switching perspectives to the implementation side, both ORM frameworks had far less code than the JDBC implementation and are easier to extend in terms of functionality. MyBatis was slightly more complex implementation-wise but still provided abstraction to decrease the amount of code required. From this point of view the three frameworks are more convenient to utilize and may be of use in projects where data structures have not been finalized and may be subject to change; or projects where data structures may be highly complex and a JDBC implementation will be too difficult to create and/or maintain.

The research questions that were defined at the start of this paper were successfully answered with data and results to substantiate. The SQL generation abilities of each candidate was discovered through studying and implementation; their implementation complexity was evaluated through analysing the amount of code required to all meet a predefined set of functional requirements; their performances were analysed through testing execution speed across a variety of queries. All these were taken into account to discuss the potential benefits they provide in software development in the previous chapter.

The time constraints present in this project coupled with an initial inexperience in utilizing the chosen persistence solutions meant that many of their aspects could not be compared. Some examples for potential future research include support for utilizing database procedures, automated schema design/creation from Java classes and additional features provided by frameworks.

Various improvements could have made to the project to make it more comprehensive. The number of queries used for performance testing could have been increased to cover more use cases such as bulk insertions, updates and deletes. Feature sets could have also been compared to provide better context for the various persistence technologies' purpose but these were not intended to have been within the scope of the project initially.

Implementation complexity can also be expanded upon by conducting experiments whereby test subjects can attempt to implement different persistence solutions individually. This will help take into account the difficulties in learning to utilize new frameworks rather than only compare the resulting lines of code.

References

[1] S. Balzer, "Contracted persistent object programming," Swiss Federal Institute of Technology Zürich, Tech. Rep., 2005.

[2] Oracle. A relational database overview. [Online]. Available: https://docs.oracle.com/javase/tutorial/jdbc/overview/database.html

[3] C. Date and H. Darwen, *A guide to the SQL standard: a user's guide to the standard database language SQL*, 4th ed. Addison-Wesly Longman Publishing], 1997.

[4] J. Gosling, B. Joy, G. L. S. Jr, and G. Bracha, *The Java Language Specification*, 3rd ed. Addison-Wesley, 2005.

[5] J. C. Process. (2016) Jsr 220 enterprise javabeans 3.0. [Online]. Available: https://jcp.org/aboutJava/communityprocess/final/jsr220/index.html

[6] E. J. O'Neil, "Object/relation mapping 2008: hibernate and the entity data model," in *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, 2008.

[7] M. Kopteff, "The usage and performance of object databases compared with orm tools in a java environment." [Online]. Available: http://www.odbms.org/wp-content/uploads/2013/11/045.01-Kopteff-The-Usage-and-Performance-of-Object-Databases-Compared-with-ORM-Tools-in-a-Java-Environment-March-2008.pdf

[8] O. Software. (2012) Jpa performance benchmark. [Online]. Available: http://www.jpab.org/OpenJPA/MySQL/server/Hibernate/MySQL/server.html

[9] JBoss. Hibernate javadoc (5.1.0 final). [Online]. Available: http://docs.jboss.org/hibernate/orm/5.1/javadocs/

[10] Apache. Openjpa documentation. [Online]. Available: https://ci.apache.org/projects/openjpa/trunk/javadoc/index.html

[11] MyBatis. Mybatis. [Online]. Available: http://www.mybatis.org/mybatis-3/