PERFORMANCE EVALUATION OF JPA BASED ORM TECHNIQUES

Neha Dhingra
Department of Electrical Engineering and Computer Science, University Of Ottawa, Ottawa, Canada

Dr. Emad Abdelmoghith
Department of Electrical Engineering and Computer Science, University Of Ottawa, Ottawa, Canada

Dr. Hussein T. Mouftah
Department of Electrical Engineering and Computer Science, University Of Ottawa, Ottawa, Canada

Abstract—Persistence and Performance are an integral part of a Java API. This paper studies the performance analysis of different JPA implementations based on the ORM framework. Large Enterprises working on JPA have become concerned about the potential pressures and overloads that a JPA implementation can endure. Java Persistence API (JPA) represents a significant role in the efficient performance of the API because the persistence process can be coordinated with the advanced features used to sustain the heavy loaded applications. Therefore, an analysis would demonstrate the impacts of different queries in various scenarios such as I/O, CPU, Garbage collection and other determinants that affect the performance of a JPA application (e.g. threads (live/Dead). Keywords: Hibernate, EclipseLink, OpenJPA, Data Nucleus, JPA

1. INTRODUCTION
Several Java API implementations such as Hibernate, EclipseLink, OpenJPA and Data Nucleus have recently begun to roll out advanced persistence technologies from their persistence providers. Many more java based companies are promising to expand into the JPA market over the next few years. Java Persistence API (JPA) has been going on for many decades experiencing periods of diminishing and growing popularity. The main advantages that have popularized java persistence have been vendor-independent programming, along with few lines of coding to persist data into the database that has adequately shifted the concern of developers to develop API based on JPAs. Now, as JDBC coding is depleted, programming costs reduced, JPA prevalence has rapidly improved, coupled with performance breakdowns, there is enough momentum for a developer to perform a performance analysis in order sustain JPA programming indefinitely in the near future. Furthermore, in today’s software market, there is increasing pressure on the developers and enterprise to adopt API that reduce the development time and programming cost in order to be considered more sustainable technology.

JPA implementations come in many variants which have lead to a myriad of acronyms that are worth summarizing. Currently, Hibernate, OpenJPA, and EclipseLink are deemed as the most successful performance oriented JPA implementations based Lazy Load concept. Although Data Nucleus is involved in using the metadata mapping through enhancers. Hibernate is not typically efficiency among the four implementation but due to its advanced features and flexibility, it is considered most scalable. Data Nucleus’s although is more advanced and prominent type of implementation on the market and is very similar to Hibernate incorporating the same components. Unlike most Hibernate APIs, Data Nucleus is efficient but involve complex persistence process. While OpenJPA is solely an apache server based implementation but lacks performance due to bug and patches issues. EclispLink APIs, which are less common, is inbuilt in JPA libraries and store as a part of the API. The focus of this study is performance analysis of the JPAs that are implemented using JPA persistence interfaces. JPA offer numerous advantages over conventional JDBC APIs such as; more efficient caching, low programming cost, less garbage collection on entities executing the SQL query, lazy loading, and less memory overhead capability for
supporting an API during the peak times. The main disadvantages are the problems while managing live threads and security in the API. However, JPA and JPA implementations are rapidly improving and through advanced features, it is expected to fall as the technology which will gain acceptance over other java based applications. The performance analysis of the JPA implementations will have a great impact on the configuration and operation of the applications. In the paper, we compare the efficiency of the JPA implementations based on numerous intricate SQL queries (e.g. Parameter Passing, Paginations, Aggregations, and Indexing). The results of this paper and approach are beneficial for evaluating the effects of a query with particular implementation and highlights reliability weaknesses which can be useful for overall smart design practices.

2. JAVA PERSISTENCE API (JPA)

In a java-based programming language, JPA is one of the most prevalent mechanisms adopted to bridge the gap between the java objects (i.e. the java classes and interfaces) and the relational databases (i.e. tables and columns). Al-though java has other mechanisms that acknowledge java applications to communicate with the relational databases such as JDBC and JDO but JPA has gained a wider adoption due to its foundations: Object Relational Mapping (ORM). ORM has obtained prevalence definitely due to it being particularly designed to perform an interaction between the object and the tables.

A typical JPA implementation includes persistence classes (e.g. EntityManagerFactory and EntityManager). EntityManagerFactory is a static class defined to perform a one-time connection configuration. An EntityManager invokes the objects of the EntityFactoryManager to persist the connection and perform the necessary Create, Read, Update and Delete (CRUD) operations. A JPA implementation comprises of the Query and Transaction objects to perform transactions and ensure a successful commit or rollback on the query. Since the metrics to measure a JPA implementation performance analysis is critical for the underlying operations performed by a JPA and the process show the effectiveness or ineffic-iveness of certain techniques based on the queries executed. So in order to obtain a concrete result, we calculated actual query performance against the relational database, the amount of time it took to execute the query is measured using a java based tool called Visual VM. One of the metrics to measure a query is the time in milliseconds it took to perform a task based on performance parameters (e.g. CPU, I/O, Garbage collection and Live Threads). Even though some JPA implementations provide metrics information by practicing configurations steps, some do not. The approach adopted to measure the performance are applied to all four JPA implementations. The time metrics for CPU and I/O are measured in milliseconds which can be skewed on the starting and ending time of the query execution.

2.1 Hibernate

Hibernate ORM framework is one of the most advanced and leading ORM frameworks. Standardizing Hibernate through JPA framework with features like indexing, batch processing and pagination optimized the output of the overall API. Hibernate JPA adds the a standardized query language called Java persistence Query language (JPQL) [10] over the inbuilt native Hibernate Query Language (HQL) [10] to further optimize the API programming. The performance analysis on hibernate is based on JPA specification, which means the implementation is based on the EntityManager and the EntityManagerFactory objects. Hibernate’s performance ranks high in contrast to its comple-ments. Furthermore, Hibernate’s equivalent to EclipseLink’s weaving and OpenJPA’s enhancement [4] processor, with a series of properties such as hibernate.default.batch.fetch size which is designed to optimize the lazy loading process in creating an optimal query and retrieve result. Another feature which improves the performance of an API in Hibernate is implemented through a new optimized ID generators that circumvent the primary key generation overhead problem. A HiLo id generator is used by default for sequences generation [4], [6], [10].

2.2 EclipseLink
EclipseLink or Eclipse persistence services project [9] is a vendor independent data persistence solution. It is a comprehensive structure which delivers persistence services by enabling developers to develop a flexible and efficient application with a wide range of feature including data formatting, object-XML binding, two-level caching structure and @converter [9] interfaces to enhance the application. EclipseLink caching strategy is very powerful and robust through co-ordinated cache for the underlying databases. Another feature of EclipseLink has been stored procedure within, it allows the user to utilize procedures and use named queries above the JPQL standards. Handling queries are the most typical task for any application through Query Hint in the Eclipselink; it addresses the issue of dynamic or named query handling and loading the join and batch efficiently. EclipseLink JPA data persistence layer supports both the relational (SQL) [8] as well as the non-relational databases (Non - SQL databases). With the introduction of composite persistence units containing the composite-persistence files working in an integrated environment, EclipseLink allows heterogeneous database integration. EclipseLink supports the most advanced and Optimized ORM services along with a complete set of interfaces and methods from the JPA specification. An EclipseLink API data persistence layer includes three major features to optimize persistence: mapped metadata (Persistent Data), shared cache (coordinate Caching Technique) [9] and query optimization using JPQL. JPA entities and plain old java object classes use weaving technique [8] to enhance the overall performance of the API [8], [9].

2.3 OpenJPA

OpenJPA is a vendor-independent implementation of the JPA specification using Apache server [7]. OpenJPA implements a monolithic java-based API for storing and retrieving of information to the relational database. It persists information through approved standard query language identified as the JPQL, that compounds with the programming methodologies of the java entity and eliminates the necessity to customize the queries for a particular database. However, JPA also supports native SQL programming, which can be used to perform quick queries with the known relational database. Open JPA is a fully compliant framework which includes the support of Java 1.5+. The data persistence features in OpenJPA such as object query language, automatic database schema, batch statement, native SQL for customized queries and two-level caching improves the performance of the API. Furthermore, OpenJPA supports Database schema automation on the application startup by setting the configuration property such as jpa.jdbc.SynchronizeMappings [7], and including jar file e.g. Openjpa-all 2.2.0 [7]. Another important property in the persistence file is openjpa.jdbc.Schema with value=“openJPA” without which entities are converted into metadata but are not persisted into the database. OpenJPA provides support for transaction handling and multiple locking mechanisms to improve the overall performance of the API. The caching layer in OpenJPA has its own implementation, which is implemented using openJPA. DataCache property.

To optimize the performance of queries from the databases to application server DataCache and QueryCache are implemented. The programming architecture of OpenJPA is fairly construed, with an EntityManagerFactory object which is invoked by the object of the EntityManager or Dispatcher Servlets using the Bean container [7].

2.4 Data Nucleus

Data Nucleus is another JPA implementation which supports data persistence either by the command line or through an integrated development environment using Eclipse IDE. It is an ORM framework, which supports transparent persistence across a wide range of data sources with a meta-data modeling through byte code enhancer and schema generation. The enhancer enhances every entities metadata before creating the schema using the "schema Tool" [13] in the data nucleus. The processes are interconnected as schema-Tool will never create database until, the Enhancer [13] performs the metadata enhancement. In DataNucleus, another important point is that the implementation is highly sensitive about the Jar file duplication even in the same workspace. The autocodeAll property in persistence.xml file creates all the entities, constraints and validation while the database is being created. To perform persistence using data nucleus it requires following jar files datanucleus-core [13], datanucleus-API-jpa-release [13], datanucleus-rdbms [13] and datanucleus-api-jdo respectively. DataNucleus as a JPA provider is a persistence framework that can save the state of the java classes and objects to the relational database to improve the performance of the API. The architecture of Data Nucleus in full compliance with JPA specification which means it implements persistence through the EntityManager and the EntityManagerFactory objects. Data
Nucleus optimizes the performance of the API through annotations or XML tags. Although XML is preferred in Data Nucleus to reduce the complexity of the persistence process, however, data nucleus supports persistence through command line programming in order to execute operations faster. A major distinction between the other JPA implementations and Data Nucleus is the built-in support for schema creation. Although data nucleus doesn’t define a mechanism to cache a query. But entity graphs [13] in data nucleus cache data through generic Compilation; they are an initial step in the Data Nucleus caching strategy because they are independent of the underlying data source and create the expression tree. The next process to cache the java entities creates a "compiled expression tree" [13], which is then converted into the native language of the relational database. Finally, the executed query returns the object cached with the "resultant" objects [13].

3. PERFORMANCE COMPARISON

From the point of view of the utility, the adoption of JPA implementation into the java application presents a number of challenges and potential reliability issues. Many practical questions are raised that have raised concern of developers such as: Will the implementation be capable of supporting the heavy load of the JPA implementation? What are the alternatives to costly SQL queries? [6] Even without JPA, many software developing enterprises are frequently managing the overloaded application. Adding JDBC's to this already fragile situation may collapse the API. JPA implementations might provide a solution to managing the persistence and performance problem which are currently under study. However, a more detailed performance analysis of the realistic SQL complex queries is required which is the aim of this paper. In JPA implementation, with the increase in the complexity of the API, the performance could be significantly stressed which would further affect the utilization of resources. Persisting data into the databases, which is the strongest in JPA implementation, could become heavily loaded thus increasing the likelihood of API to failure, that potentially affecting the overall performance of the application. The proliferation of JPAs into the java programming may cause a number of undesirable side effects such as lack of proper support for functionalities including stored procedures, Indexes, and the new querying strategy that could create memory leaks. All of these impacts and are studied in this paper through a performance comparison test based on five complex SQL queries. JPA implementation persists data into the databases and then retrieve that data to execute the SQL operations. This paper focuses performance analysis on the latter. Therefore, to better understand the impacts of multiple JPA implementation through compli-cated SQL queries, extensive implementations were created in all four JPA implementations for a realistic JPA functions. The studied JPAs are highly detailed and consist of the performance criteria to measure the CPU usage, I/O usage, Garbage collection and Threads consumed by a particular query [2]. This study is unique in which both queries and performance parameters are integrated and analyzed simultaneously with only focus on one type implementation at one time. Based on performance analysis, comparisons are made for different JPA implementation using queries [2], [6].

3.1 CPU and Memory Usage

In order to accurately evaluate the impacts and stresses of the query on an implementation, a realistic console based application was developed. Unlike existing analyzes, the impacts on the query based on CPU and I/O were measured to include the number of CPU cycle required for a particular query to execute (Fig. 1). Core Processing Unit (CPU) is the brain of any Java Virtual Machine (JVM)-based application. A higher rate of CPU usage in a java based application means that the API is degrading in performance. While a medium and low reading are preferred per CPU cycle. The CPU usage in a JPA implementation is measured in percentage between the scale of 0-80 percent. A significant amount of CPU cycle is needed for a query to execute and demonstrate the output on a console. A developer can monitor and manage the queries performance for any java based application through Visual VM. To simply the process Visual VM monitor the same process and visually displays the output. Each implementation consists of the percentage of CPU usage representing the SQL queries and high usage. In the paper highest CPU have been recorded in OpenJPA. Out of five queries, OpenJPA’s count query consumed CPU below 40 percent. While other queries like aggregation, pagination, inner join and parameter passing affected the performance by catching the usage rate above 40 percent. Furthermore, Hibernate and Data Nucleus included smaller variation in performing the query and retrieving the result between 1 percent to less than 20 percent whereas 20 percent to 30 percent in DataNucleus. The remaining implementation i.e. EclipseLink represented the lumped loads from 4 percent to
15 percent including a rise in the pagination query to 60 percent. The total percentage of all the five queries were averaged and represented in Fig. 1.

Throughout this evaluation phase, Hibernate and DataNucleus outperform other implementations. Memory in JVM consists of 3 major segments mainly, the heap, the metaheap and the miscellaneous. Heap memory store the java objects that at run-time allocates data area to execute start up processes. By default every heap size vary according to the implementations and number of classes loaded in the query. For any programmer knowledge of heap, dealing with the heap space, analyzing the heap dumps is very important. When a java program starts JVM it retrieves some memory from the Operating System (OS) or when an object is created using a new operator or object is allocated memory all the processes are executed through heap. Default heap size is half of the physical memory. In a JPA implementation measuring initial size of the heap and the maximum units it acquired at the end of a query affects the performance of the application. Though Visual VM we can monitor a GUI format which is easy to interpret and evaluate while comparing all 4 implementation. In the graph below Fig. 2 and Fig. 3, it shows the average of heap size and units consumed by 5 different queries. The size of the heap fluctuates with every execution wherein the number of processes in the memory are management in the background. In any JPA implementation classes are the most significant part of the application. Mapping process in an application includes the java objects or classes and the database relations and column. When an application is loaded whether to just create a schema or to execute a simple or complex queries it loads classes to create the object and perform operations. Fig. 2 and Fig. 3 indicate the Heap size and Head used by entities of all JPA implementations for selected queries. For a high number of classes the heap size and the number of heaps used, significant overloads the API. JPA overloads are not uncommon and are often tolerated for peak times, especially for any java application, however, it would be more important aspect to understand the the size of a heap along with the used heaps for the particular query.

3.2 Threads Count (Live/Daemon Threads)
A typical JPA implementations increase/decrease the load in an API based on the number of threads (Live/Daemon Threads). Threads are lightweight processes which perform simultaneously several task in the background. These operations are either synchronous or asynchronous. In any Java based application thread performs some of the majors
performance oriented interprocess communications. In a JPA implementation threads represent the major role in determining fetching technique while executing a query. All 4 JPA implementation load processes either in lazy or eager mode (Fig. 4 and Fig. 5)

![Live Threads](image1)

**Fig. 4: Metrics of Live Thread in 4 JPA implementations performing 5 SQL complex Queries**

![Daemon Threads](image2)

**Fig. 5: Metrics of Daemon Thread in 4 JPA implementations performing 5 SQL complex Queries**

with the scale distribution between 0-12 threads in Visual VM. For determining the loads realistically, the live and daemon threads were profiled. In this study, the performance analysis on thread count is selected because it is expected that the higher end of thread count would be more affordable and will likely to improve the performance initially by avoiding the problem of the thread deadlock and thread pool configuration issues. Although all the implementations performed similarly in the case of Hibernate, OpenJPA and Data nucleus to executing an inner joins and aggregation with a thread count between 8.5 to 11 in Live threads and 8.5 to 10 in Daemon Threads. A Daemon thread is a thread that does not prevent the java virtual machine from exiting when the program concludes. Hibernate has a higher thread count enough to avoid deadlock and improve the concurrency control in persisting objects and queries of varying sizes and complexity in a reasonable measure. However, the limitations of JPA must be considered. A standard JPA implementation over ambitiously caches every object, entity and query which influences the performance of the API.

Findings indicate that the Hibernate and OpenJPA implementation thread count has a significant improvement while impacting on the API. A possible case is shown in Fig. 4, 5 while executing aggregation query. Depending on the thread count of a JPA implementation, the API load reaches peak that significantly influenced the performance of the API. Results indicate that the efficiency of a JPA implementations can be significantly impacted by the number of threads executing in a query to retrieve a result from the database.

### 3.3 Number of Classes Loaded

Given the JPA implementation introduced in section 2, it can be deduced that performance analysis of the java API would improve the efficiency of the application and generally the queries that are executed to retrieve the data from the database. The concept of measuring the number of classes that are loaded is appropriate due to the complexity of the API and faster development in correlation with additional software emerging technologies and peculiarities. Analyzing the performance in term of the number of classes loaded would provide the developer with the high-level knowledge of the internal architecture of the JPA. Visual VM represents a criteria to measure the number of classes loaded in executing a particular query and demonstrates the total number of objects allotted by each class (including array classes) in a relation. The total number of classes that are loaded are represented by the percentage of bytes as well as the number of bytes allocated to each class. In a JPA based API entities are the significant part of the application. For each entity loaded in order to execute the SQL operation currently in the JVM, the profiler issues
count on the size and number of objects designated since the profiling session started. The outputs are automatically refreshed as new objects are earmarked and as new classes are loaded. For any java application, it would be an important aspect to understand the number of classes that are loaded for the particular query. Visual VM is a tool the provides a graphical display of the information which is them averaged (Fig. 6).

![Total Loaded Classes](image)

**Fig. 6: Metrics of Loaded Classes in 4 JPA implementations performing 5 SQL complex Queries**

It describes the visual representation of the overall loaded classes with the number of shared classes in a single application through the scale of 0-6000 classes. Although Hibernate consumed the maximum number of classes in order to perform operation still the performance of Hibernate is optimal when compared with other JPA implementation. OpenJPA rather consumed less number of classes while performing the complex SQL operation (between 3500-4500 bytes) but it faced many trivial problems. Although Data Nucleus outperform all the JPA implementation (between 3200-3600 loaded classes on an average) it is not adopted as an API due to high complexity.

### 3.4 Garbage collection

In a JPA garbage collection is a process through which a processor eliminates the dead objects from the java heap space and yields memory back to the heap. The method is automatically implemented in all the JPA frameworks. But garbage collection in all 4 implementations exhibited distinctive result in correspondence with the complexity of the SQL queries such as the inner join, aggregation, association, pagination, and count. In JPA, the JVM is a distributed heap that is divided into three segments sordid on the characteristics of data the permanent data, the old data and the live/active data. New objects generate a heap allocate in the active domain. During the execution of the query many objects are alive or dead, however, objects that reside alive subsequently after the operation transit to a permanent location whereas the dead objects that are assembled into garbage collector are re-initialized. The permanent space in the JVM stores metadata about classes and methods that are persisted. In graph (Fig. 7)

![Garbage Collection](image)

**Fig. 7: Metrics of Garbage collection in 4 JPA implementations performing 5 SQL complex Queries**

It depicts the average mean of five SQL queries performed on four different JPA implementation. Hibernate performed enormously with less then zero percent garbage collection in few queries while OpenJPA and Data Nucleus lacks performance orientated result due to heavy memory leaks especially pagination and count query in later.

### 1. CONCLUSION

The goal of this paper was to highlight potential performance issues for a realistic JPA based implementation to measure the performance of the complex SQL queries. The impact of different JPA implementations, based on performance parameters, and existing load variations using five SQL queries was examined using Eclipse IDE for entities and Oracle 12g for persisting data into the relational database. The main conclusions of this paper are as follows:

- Hibernate and EclipseLink JPA has a significant positive impact on the performance in terms of CPU and Memory usages, managing threads and garbage collection even
while executing the most complicated query in the peak demand.
- OpenJPAs performance is most significantly impacted while executing queries and the compatibility issues have further added constraints with the Java 1.6+ version.

Although Data Nucleus has shown significant signs of performance oriented JPA implementation. But the JPA implementation lacks in popularity due to heavy complexity to perform the persistence process. JPA make interfaces and JPA implementations are dependent on each other. This relationship provides a new opportunity for developers persist the java application through minimal code. With development of JPA, employing java applications as persistence process would be more strengthened and desirable for an API. From this point of view, a developer the performance analysis would further provide a concrete result based on the JPA utilities and leads to a more satisfaction for an enterprise using JPA. Due to the existing challenges in software development, based on unwanted loads, and development of low costly software it is essential to assert analysis on the technologies and obtain preferences through properly detailed documentation.

5. FUTURE WORK

In future a unified set of features for all JPA specifications for easy migration among the JPA implementations would be useful. In terms of optimizing Query Language in Data nucleus, the implementation requires more manageability through its own transaction management system. OpenJPA, on the other hand, requires more flexibility in terms of handling complex queries and data-type and a compatibility support for Java 1.7+ version. We need a more planned future comparative studies for tool based and new features based, including a new persistence cache system. Also, this comparative study is prepared practicing four implementations and five queries in Oracle12g DBMS. It might be fascinating to see how these tools perform using other open source databases management system such as MySQL and SQLite. Although the results presented in EclipseLink were difficult to retrieve there is a need for more stability in the implementation because of high crashing rate in visual VM. Another future approach would be comparing commit and rollback architecture in Data nucleus, so in order to assert how deadlocks are managed in a JPA application.

REFERENCES


