Analysis of ORM based JPA Implementations

by

Neha Dhingra

Thesis submitted to the
In partial fulfillment of the requirements
For the MCS Degree in
Computer Science

School of Electrical Engineering and Computer Science
Faculty of Engineering
University of Ottawa

Abstract

Large scale application design and development involve some critical decisions, and one of the most important issues that affect software application design and development is the technology stack used to develop an extensive system. In an JPA API, response time is often a measure of how quickly an interactive system responds to user input. Persisting framework, such as Object Relational Mapping (ORM) are applied to manage communications between an object model and data model components and are vital for such systems. Hibernate is considered the most effective ORM framework due to its advanced features, and it is the de-facto standard for Java Persistence API (JPA)-based data persistent frameworks. This thesis comprises a review of the most widely used JPA providers, particularly frameworks that provide JPA support such as Hibernate JPA, EclipseLink, OpenJPA and DataNucleus JPA.

In current java programming, APIs based on persistence and performance are integral aspects of an application. Performance analysis of the above four JPA implementations is based on the ORM framework that contributed most significantly to discovering the challenges and verified the programming considerations in the language. For a large-scale enterprise, working on JPA is always tedious due to the potential pressures and overloads of the implementations, as well as the comprehensive guarantee, that is required while adopting the technology. A JPA implementation continually persists data into the database at runtime, by managing persistence processes through interfaces and classes, that often needs optimization, to provide performance-oriented results at heavy loads. Therefore, in this thesis a detail feature analysis was performed, before the performance analysis. To enhance the comparison of the persistence framework, an extended experiment with a cloud database using Database-as-a-service (DBaaS) versus Physical Persistence was performed, using a comparative approach for all four JPA implementations. Different SQL queries on cloud versus physical persistence for JPA applications were measured using CPU, GC, and threads (live and daemon). Finally, a statistical analysis was performed using the Pearson's correlation coefficient and a steady/start-up phase.

Acknowledgements

My sincere gratitude to Professor. H. T. Mouftah for providing me with guidance, support, advice and ideas throughout this work. I whole heartedly thank him for his valuable supervision.

My deepest appreciations to my mother and my entire family for their unconditional love and support during my time here. A special word of thanks to my parents, family and friends for helping me keep my motivation.

Finally, I would like to thank all my teachers and mentors who have been instrumental in providing me with the capability to pursue my Masters degree.

Table of Contents

I Abstract ii
II Acknowledment iii
III Table of Contents vi
IV List of Tables ix
V List of Figures x

1 Introduction **1**
1.1 Motivation 2
1.2 Objective 4
1.3 Contributions 6
1.4 Thesis outline 6
1.5 List of Publications 7

2 Literature Review **8**
2.1 JPA implementation Design and Operations 9
2.2 Introduction to ORM based JPA Providers 10
2.2.1 Design and Model of Hibernate JPA implementation 11
2.2.2 Design and Model of EclipseLink JPA implementation 12
2.2.3 Design and Model of OpenJPA implementation 14
2.2.4 Design and Model of Data Nucleus JPA implementation 17
2.3 Comparison of ORM based JPA implementations 18
2.3.1 Query and transaction 18
2.3.2 JPA Object classes 19
2.3.3 Mapping and Distributed Database unit 19
2.3.4 Auto-insert 20
2.3.5 Connection and configuration 20
2.3.6 Performance optimization Features 21
2.3.7 Cache 21

2.3.8 Threading 21
2.4 Conclusion 27

3 Performance Analysis of JPA implementations of 5 major Queries 28
3.1 Persistence Platforms 29
3.2 Introduction to Performance Comparison 30
3.2.1 Hibernate JPA Performance Oriented Implemetation 32
3.2.2 EclipseLink JPA Performance Oriented Implemetation 33
3.2.3 OpenJPA Performance Oriented Implemetation 34
3.2.4 DataNucleus JPA Performance Oriented Implemetation 35
3.3 Performance Analysis of JPA implementations Physical 36
3.3.1 CPU and Memory Usage 37
3.3.2 Threads Count (Live/Daemon Threads) 38
3.3.3 Number of Classes Loaded 40
3.3.4 Garbage collection 41
3.4 Conclusion 41

4 Review and Analysis of JPA based implementation on Cloud 42
4.3 Analysis Result 43
4.3.1 Hibernate JPA Local versus Cloud database 44
4.3.2 EclipseLink JPA Local versus Cloud database 45
4.3.3 OpenJPA Physical Local versus Cloud database 45
4.3.4 DataNucleus JPA Local versus Cloud database 46
4.4 Experiment Results 54
4.5 Conclusion 60

5 Statistical Review and Analysis of JPA based implementation 61
5.1 Challenges and motivation Towards Analysis 62
5.2 JPA Frameworks 63
5.3 Experiment Design 64
5.3.1 Evaluation of JPA frameworks 65
5.3.2 Benchmarks, Machine, Hardware and Software 68
5.3.3 Event Execution with Start-up Rate and Numerical Result 73
5.3.4 Event Execution with steady-up Rate and Numerical Result . . . 76
5.4 Final Analysis 78
5.4.1 Final Start-up Result 80

5.4.2 Final Steady Result 84
5.5 Conclusion 86

6 Conclusion and Future Work **87**
6.1 Conclusion Remarks 87
6.2 Future work 88

Appendix A Confidence Interval **95**

Appendix B Statistical Experiment Results **97**

Appendix C Query Results **102**

Appendix D Results of JPA Implementations using Number of classes and Thread counts (Live/Daemon) in Local versus Cloud Database **118**
D.1 Analysis of JPA Framework 119
D.1.1 Hibernate JPA Local versus Cloud Database 119
D.1.2 EclipseLink JPA Local versus Cloud Database 120
D.1.3 OpenJPA Local versus Cloud database 121
D.1.4 DataNucleus JPA Local versus Cloud database 123

List of Tables

Table2.1 Query and Transaction in JPA 23
Table2.2 Object classes in JPA 24
Table2.3 Mapping And Distributed Database unit in JPA 24
Table2.4 Auto-Insert in JPA 25
Table2.5 Connection and configuration In JPA 25
Table2.6 Performance Optimization Features in JPA 26
Table2.7 Cache in JPA 27
Table2.8 Threading In JPA 27

Table4.1 Feature Comparison of DBaaS (Oracle Versus Amazon) Part 1 . . . 47
Table4.2 Feature Comparison of DBaaS (Oracle Versus Amazon) Part 2 . . . 48
Table4.3 Data Entries for Query 1 58
Table4.4 Response Time Values of Different Entries for Local and Cloud Database in Query 1 58
Table4.5 Data Entries for Query 2 58
Table4.6 Response Time Values of Different Entries for Local and Cloud Database in Query 2 58
Table4.7 Data Entries for Query 3 59
Table4.8 Response Time Values of Different Entries for Local and Cloud Database in Query 3 59
Table4.9 Data Entries for Query 4 59
Table4.10 Response Time Values of Different Entries for Local and Cloud Database in Query 4 59

Table5.3 Hibernate JPA Benchmarks 69
Table5.4 EclipseLink JPA Benchmarks 71
Table5.5 OpenJPA Benchmarks 72
Table5.6 DataNucleus JPA Benchmarks 73

Table5.7 Pearson's Correlation coefficient results for Hibernate JPA with positive correlation between JPA Benchmarks 77
Table5.8 Pearson's Correlation coefficient results for EclipseLink JPA with positive correlation between JPA Benchmarks 77
Table5.9 Pearson's Correlation coefficient results for OpenJPA with positive correlation between JPA Benchmarks 77
Table5.10 Pearson's Correlation coefficient results for DataNucleus JPA with positive correlation between JPA Benchmarks 78
TableB.1 Average Mean calculating GC cycle over JPA implementation using 30 execution per JPA Benchmarks per SQL query 98
TableB.2 Average Mean calculating CPU cycle over JPA implementation using 30 execution per JPA Benchmarks per SQL query 99
TableB.3 Standard Deviation of GC cycle over JPA implementation using 30 execution per JPA Benchmarks per SQL query 100
TableB.4 Standard Deviation of CPU cycle over JPA implementation using 30 execution per JPA Benchmarks per SQL query 101
TableC.1 Data Entries for Query 5 112
TableC.2 Response Time Values of Different Entries for Local and Cloud Database in Query 5 112
TableC.3 Data Entries for Query 6 112
TableC.4 Response Time Values of Different Entries for Local and Cloud Database in Query 6 112
TableC.5 Data Entries for Query 7 113
TableC.6 Response Time Values of Different Entries for Local and Cloud Database in Query 7 74
TableC.7 Data Entries for Query 8 113
TableC.8 Response Time Values of Different Entries for Local and Cloud Database in Query 8 113
TableC.9 Data Entries for Query 9 113
TableC.10 Response Time Values of Different Entries for Local and Cloud Database in Query 9 114
TableC.11 Data Entries for Query 10 114
TableC.12 Response Time Values of Different Entries for Local and Cloud Database in Query 10 114
TableC.13 Data Entries for Query 11 114

TableC.14 Response Time Values of Different Entries for Local and Cloud Database in Query 11 114
TableC.15 Data Entries for Query 12 115
TableC.16 Response Time Values of Different Entries for Local and Cloud Database in Query 12 115
TableC.17 Data Entries for Query 13 115
TableC.18 Response Time Values of Different Entries for Local and Cloud Database in Query 13 115
TableC.19 Data Entries for Query 14 115
TableC.20 Response Time Values of Different Entries for Local and Cloud Database in Query 14 116
TableC.21 Data Entries for Query 15 116
TableC.22 Response Time Values of Different Entries for Local and Cloud Database in Query 15 116
TableC.23 Data Entries for Query 16 116
TableC.24 Response Time Values of Different Entries for Local and Cloud Database in Query 16 116

List of Figures

Figure2.1 Three Tier Architecture with JPA mapping layer 10
Figure2.2 Hibernate JPA High level Architecture 11
Figure2.3 EclipseLink JPA High level Architecture 14
Figure2.4 OpenJPA High level Architecture 16
Figure2.5 DataNucleus JPA High level Architecture 17

Figure3.2 Metrics of CPU Usage in 4 JPA implementations performing 5 SQL complex Queries 34
Figure3.3 Metrics of Memory Size in 4 JPA implementations performing 5 SQL complex Queries 35
Figure3.4 Metrics of Memory Used in 4 JPA implementations performing 5 SQL complex Queries 38
Figure3.5 Metrics of Daemon Thread in 4 JPA implementations performing 5 SQL complex Queries 38
Figure3.6 Metrics of Live Thread in 4 JPA implementations performing 5 SQL complex Queries 39
Figure3.7 Metrics of Number of classes Loaded in 4 JPA implementations performing 5 SQL complex Queries 40
Figure3.8 Metrics of Garbage Collection in 4 JPA implementations performing 5 SQL complex Queries 41

Figure4.1 JPA implementation using cloud database with Database-as-a-service (DBaaS) 43
Figure4.2 Cloud Computing Architecture 44
Figure4.3 Number of CPU and GC cycle in Hibernate JPA using local versus cloud database 48
Figure4.4 Heap used and Heap Size in Hibernate JPA using local versus cloud database 49

Figure4.5 Number of CPU and GC cycle in EclipseLink JPA using local versus cloud database 49
Figure4.6 Heap used and Heap Size in EclipseLink JPA using local versus cloud database 50
Figure4.7 Number of CPU and GC cycle in OpenJPA using local versus cloud database 51
Figure4.8 Heap used and Heap Size in OpenJPA using local versus cloud database 52
Figure4.9 Number of CPU and GC cycle in DataNucleus JPA using local versus cloud database 53
Figure4.10 Heap used and Heap Size in DataNucleus JPA using local versus cloud database 54

Figure5.1 Hibernate JPA Persistence.XML file 65
Figure5.2 EclipseLink JPA Persistence.XML file 66
Figure5.3 OpenJPA JPA Persistence.XML 67
Figure5.4 DataNucleus Persistence.XML 68
Figure5.5 First stage (Start phase) results using CPU cycle as scaling factors for JPA benchmarks (fetch size per increasing batch size) on y-axis and JPA implementation on X-axis 75
Figure5.6 First stage (Start phase) results using GC cycle as scaling factors for JPA benchmarks (fetch size per increasing batch size) on y-axis and JPA implementation on X-axis 75
Figure5.7 First stage (Steady phase) results using CPU cycle as scaling factors for JPA benchmarks (fetch size per increasing batch size) on y-axis and JPA implementation on X-axis 76
Figure5.8 First stage (Steady phase) results using GC cycle as scaling factors for JPA benchmarks (fetch size per increasing batch size) on y-axis and JPA implementation on X-axis 76
Figure5.9 Second stage (Start phase) Mean of 30 iteration with 95% confidence in results per JPA benchmarks with CPU cycle on y-axis and increasing JPA Benchmarks on X-axis 82
Figure5.10 Second stage (Start phase) Mean of 30 iteration with 95% confidence in results per JPA benchmarks with GC cycle on y-axis and increasing JPA Benchmarks on X-axis 82

Figure5.11 Second stage (Steady phase) Mean of 30 iteration with 95% confidence in results per JPA benchmarks with CPU cycle on y-axis and increasing JPA Benchmarks on X-axis 83
Figure5.12 Second stage (Steady phase) Mean of 30 iteration with 95% confidence in results per JPA benchmarks with GC cycle on y-axis and increasing JPA Benchmarks on X-axis 83
Figure5.13 Pearson's Correlation coefficient Result for Hibernate JPA with positive correlation between JPA Benchmarks 84
Figure5.14 Pearson's Correlation coefficient Result for EclipseLink JPA with positive correlation between JPA Benchmarks 84
Figure5.15 Pearson's Correlation coefficient Result for OpenJPA with positive correlation between JPA Benchmarks 85
Figure5.16 Pearson's Correlation coefficient Result for DataNucleus JPA with positive correlation between JPA Benchmarks 85

FigureD.1 Number of classes called in Hibernate JPA using local versus cloud database 119
FigureD.2 Number of Thread (Live/Daemon) in Hibernate JPA using local versus cloud database 120
FigureD.3 Number of classes called in EclipseLink JPA using local versus cloud database 121
FigureD.4 Number of Thread (Live/Daemon) in EclipseLink JPA using local versus cloud database 122
FigureD.5 Number of classes called in OpenJPA using local versus cloud database 122
FigureD.6 Number of Thread (Live/Daemon) in OpenJPA using local versus cloud database 123
FigureD.7 Number of classes called in DataNucleus JPA using local versus cloud database 123
FigureD.8 Number of Thread (Live/Daemon) in DataNucleus JPA using local versus cloud database 124

Chapter 1

Introduction

Object/Relational Mapping (ORM) is a technique to dynamically transform data from an object-oriented model into a relational database (RDBMS) model [2] at run-time. A typical Object-Oriented Program (OOP) manages entities and classes, while relational database management systems are based on relations, and map two Architecture (JPA and JPA implementations) based on ORM framework through a JPA mapping layer, and reduce complex boilerplate code. With only a few lines of code (LOC), this programming style can shift a developer's focus to building a high-end optimized JPA application with less code, and avoid programming delays.

Building a JPA application with advanced methods and interfaces using JPA repositories improves the persistence process, as well as advanced JPA features such as Two-level Caching and DB Schema generation improves the JPA API. Although persistent, JPA API developed in a physical environment required less management, and it was noted that in a broad spectrum there are other persisting platforms on the internet, known as cloud computing services or Database-as-a-service (DBaaS). These can be private, public or hybrid clouds, Chapter 4 include a feature comparison of two major cloud providers, using factors such as privacy and security. With 90% of the businesses [1] moving towards using clouds for database persistence, the intention was to measure the performance of JPA API using local versus cloud database. A CUI application was built to measure the performance of the JPA API using DBaaS, to analyze the number of CPU cycles, the GC collection performed, the number of classes called in the execution, and the number of live/daemon threads required to perform database access operations, including create, read, update and delete (CRUD) queries.

It was believed that migrating JPA implementations on cloud databases would provide

the capability to manage and monitor JPA persisted databases online with reduced cost and advanced functionality. JPA API persistence on a local system was more optimal when executing CRUD operations compared well to DBaaS on the cloud. Finally, a rigorous statistical analysis using Pearson's correlation coefficient was performed to analyze the correlation between various JPA benchmarks, using a local database.

1.1 Motivation

Since the introduction of open stack JPA frameworks, java developers have been working toward building a standard JPA API from the set of java classes and interfaces. With several open and commercial JPA providers in the market, building a JPA API provides ease and portability in the code. The JPA persisted database can be accessing and stored on a local system, thereby eliminating the need to manage mismatches between Java classes and POJOs [3]. A new persistence platform that is independent of the underlying architecture avoids overhead due to physical disks, allowing the JPA API to persist data on a cloud through DBaaS. To understand and compare the JPA persistence platforms, a performance analysis based on CRUD operations, which includes JPA implementations and Oracle 12 c database on the cloud as Database-as-a-Service (DBaaS) was performed comparing CPU cycles, GC collection, and the number of Live/Daemon Threads. For database persistence of the JPA implementations, potential providers included Oracle 12 cloud services and Amazon cloud services. One factor that required consideration was the optimal performance to capture real-time scenarios in a JPA API. The verification process was based on major parameters, namely CPU, Garbage Collection (GC), Threads, Number of classes Loaded and Memory Heap. Thus, the execution used a comparative analysis of the JPA implementation, using physical database persistence versus DBaaS.

1.2 Objective

The objective of the thesis was to identify, which existing JPA implementation would be most advantageous in terms of features, and to ensure potential persistence benefits using local and cloud databases. The features reviewed and addressed in the thesis are based on the performance, metadata mapping and connection strings in the JPA providers. With respect to the design itself, the main objective was to keep the implementations lightweight and simple so that JPA APIs use dynamic mapping mechanisms, and fill the persistence gaps in the platform with corrective features [31]. To demonstrate the

performance of the proposed JPA implementations using CRUD operations, the goal of the JPA API was to obtain results from potential areas of concern proposed by a typical software developer, including CPU, GC, and Threads. The major research question of the thesis was:

- Analyze the JPA implementation using cloud database versus physical databases, in terms of response time, using verification parameters such as CPU utilization, Garbage collection, and Number of threads called.
- To build a JPA implementation performance analysis with respect to the features such as scalability, security, and availability while persisting data on the database.

1.3 Contributions

The JPA API performance analysis was initially created in the local database using Oracle 11g as persisting platform followed by an upgrade on to the Oracle 12c. The next step was to migrate and deploy the existing JPA API onto cloud databases, using Oracle cloud services or Amazon cloud services. A comparison between the JPA API database persistence platform was created to indicate response times, CPU utilization, Number of Live thread and Garbage collection. The main research contributions of this thesis are as follow:

1. Provided better feature analysis for existing JPA implementations to minimize the complexity for native developers.
2. Developed an efficient performance analysis which performed complex JOIN, SUBJOINS and AGGREGATIONS using local persistence verifying parameters such as the number of CPU and GC cycles, number of classes called and the number of Live/Daemon threads.
3. Proposed an original performance analysis experiment to evaluate JPA implementations using the physical versus cloud database with following parameters:
   - Average CPU cycle (percentage).
   - Average GC (percentage).
   - Average of Memory Heap Use /Heap Size

- Average Number of Threads (Live / Daemon)
- Average Number of classes Loaded in executions.

4. Provided a statistical analysis using JPA benchmarks to extract and evaluate Pearson's correlation coefficient over the steady period.

1.4 Thesis outline

The remainder of this thesis is organized as follows. In Chapter 2, a Literature Review was performed on existing JPA implementations, in particular, JPA features. Since the features analysis of JPA providers is an active research area, the focus was to compare JPA features, and briefly discuss their JPA providers architecture .
In Chapter 3, a performance analysis was performed on four different JPA implementations, and it was presented using an experimental design. It was a novel approach in the existing application, developed in JPA implementations with advanced feature to capture the performance of each implementation. The feature analysis in the previous section was used to demonstrate the improved effectiveness of JPA using these features in the presence of complex SQL operations.
Chapter 4, proposed a new persistence platform for JPA implementations, employing database-as-a-service with a cloud. A set of features, challenges, and advantages (Table 4.1 and 4.2) of adding cloud database into JPA implementations, as well as performance comparisons with the physical persisting platform, were analyzed.
In Chapter 5, a statistical analysis using JPA implementations, and Oracle 12 c with Visual VM and JCT framework, was conducted using a JPA benchmark to compare the start-up and steady states. The steady performance was analyzed with Pearson's correlation coefficient. Chapter 6, discusses the contributions, results, and recommendations, and proposes ideas for future research.

1.5 List of Publications

1. N. Dhingra, E. Abdelmoghith and H. T. Mouftah. Review on JPA Based ORM Data Persistence Framework, ICECS 2016 Toronto, Canada. IJCTE, 9th International Conference on Environmental and Computer Science. 2016.
2. N. Dhingra, E. Abdelmoghith and H. T. Mouftah. Performance comparison of ORM based JPA implementations, Proceedings of 2nd International Conference on

Computer Science Networks and Information Technology, in Montreal, Canada, ISBN: 978819313736915, 2016.
3. N. Dhingra, H.T.Mouftah, "Statistical Analysis of ORM based JPA implementations physical database", IET Software Journal (Under Review) , 2017.

Chapter 2

Literature Review

To preclude mapping concerns, ORM bridges the gap between the platforms and manages the disparities between the object graphs and the structured query language (SQL). For a developer, segregated mapping layers depreciate the complexity of the boilerplate code. ORM combines the functionality of conventional Java Database Connectivity (JDBC) programming models [3] and the persisted databases. A conventional ORM application yields a lightweight object-oriented interface known as a Data Access Object (DAO) [13]. A DAO layer determines the design pattern that adds Java entities to a sequence of SQL operations (e.g. insert, delete, update) through predetermined functions. To execute a query and retrieve the relational data efficiently in the object-oriented programming, DQL (Doctrine Query Language) [4] was introduced to reduce user complexity to simple data definition language (DDL) commands. DQL is a platform to retrieve the Java entities using predetermined sets of protocols. Apart from the programming convention, ORM accelerates optimization by transaction locking and maintains data writes through defined transactional boundaries [2, 3, 11]. ORM attunes data accessed in a record-based pattern and standardizes the persistence process through the Java Persistence API (JPA) interface. JPA is a Java application programming interface [3] that manages data between Java objects and relational databases. It is a specification, not implementation to persist data in the RDBMS. Due to the failure of the enterprise persistence model, and lack of Java persistence standards, developers often materialize Hibernate JPA implementations to optimize the mapping architecture [16]. Exiting JPA implementations increase portability and extensibility of the code, by decoupling the JPA specifications from the underlying JPA API architecture. Following sections discuss a comparison based on existing JPA implementation using features that optimize an application's performance.

2.1 JPA Implementations Design and Operations

JPA is an empty interface that persists java classes, and interfaces with relational database tables and columns. To perform the mapping, a typical JPA specification employs a set of predefined methods and interfaces, which provide methodologies and standardized abstract programming using ORM [12]. This high-level architecture is divided into java classes and databases, which provides a bridge between dissimilar platforms through the ORM mapping framework. Hibernate, the most advanced JPA implementation, was developed by JBOSS, RedHat [5], EclipseLink JPA by Oracle and SUN glass [7], Open JPA by IBM and Bea [6] and Data Nucleus by JPOX and Tapestry [20] are commercially available, vendor independent providers that follow the JPA paradigm to configure an API.

According to the author [3], JPA is a standard-compliant framework designed for mapping Plain Old Java Object (POJO) into relational databases. To achieve this, JPA implementations perform metadata modeling and schema creation that produces applications through standard annotations, by defining a @annotation name or with the XML files as tags. Annotations are preferred standard than XML, due to simplicity and ease in handling injections compared to tags. Both mapping techniques follow similar functionality, but due to its high complexity XML is not preferred for the programming paradigm. For example, an annotation/XML tag to create a primary key column in an entity is defined by an @ID annotation, and the XML syntax is: < name = "ID" value = "integer" > . A typical JPA application defines run-time interface objects to persist data and create a connection with the database. To create connections between the Java objects and the database, JPA defines an EntityManagerFactory interface object [18], which allocates and deallocates resources. Once the mapping process is complete the resources are detached and the manager is destroyed. An EntityManagerFactory object is invoked by the object of an EntityManager interface [18], to start the persistence process. The EntityManager then interacts in a persistence context to perform entity create, read, update and delete (CRUD) operations. With JPA persistence, the SQL operations are managed using transaction and query objects, which retrieve and execute Data Definition Language (DDL) operations. Figure 2.1 shows a three layer high-level architecture, where the top layer is the Graphical User Interface (GUI) that communicates with the client/server to perform operations at the front-end. The middle layer is the JPA layer, which is sub-divided into a service layer using controllers, DAO, repositories and service implementation, and a data persistence layer that defines the mapping procedure between

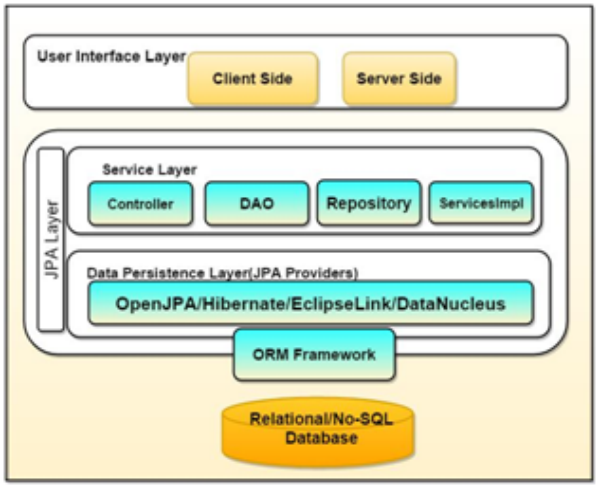

Figure 2.1: Three Tier Architecture of a Typical JPA API with JPA as the Middle Layer for Mapping

the relational database and Java entities. The bottom layer determines the type of RDBMS to persist data in a tabular format.

2.2 Introduction to ORM based JPA Providers

2.2.1 Design and Model of Hibernate JPA Implementation

Hibernate, the vendor-independent ORM framework, maps a java object-oriented model into a relational database, accessing both the POJOs and the database. According to B. Vasavi et. al [17], Hibernate provides high performance, feature rich mapping technology to persist java data types into the underlying structured query language (SQL) data types. For example, an integer field in java class is converted into an int (integer) column in the SQL server. As a comprehensive solution, Hibernate achieves persistence not only by managing java entity mapping to the database tables but also overcoming the development time wasted on binding a JPA API through a java API and database. A study by B. S. Sapre et. al in [15] found that to avoid the impedance mismatch problem, Hibernate mapping processes use run-time persistence properties to create an association between java classes and relational fields. Because Hibernate is free it is distributed under the GNU Lesser General Public License 2.1, to provide open source access to developers [5]. The Hibernate mapping process is based on the lazy/eager mode; that is, all necessary information about the classes, such as generating the schema,

creating stubs for java sources and creating primary-foreign key relationships between the entities, is loaded at runtime or compile time. Similarly, Hibernate documentation [5], indicates it follows high-level abstraction by encapsulating the underlying architecture of the API from the developer, which diminishes the complexity of the code. This process increases the extensibility of the code but limits the developer to abstract methods. A study by L. Semebera. in [18] stated that Hibernate JPA provides a detailed assessment of queries accessing the database through Hibernate Query Language (HQL). HQL is an advanced query language used to execute CRUD operations from the Java classes in and out of a database. The configuration object in Hibernate JPA, comprised of the properties about the connection to be configured. The configuration object addresses and authenticates Hibernate properties by creating a connection to the local database. A configuration object is used as the SessionFactoryManager to start the persistence process and create the connection between java entities and the database tables. The SessionFactoryManager object is then invoked by the session instance to execute SQL operations. In this process, the session object maintains the cache and requires an explicit Close () method to avoid memory leaks [5]. To perform transactions on the database, a session object in Hibernate API is used to invoke the transaction object, which then executes operations in a session through the transaction manager and the underlying query. [31] stated that Hibernate allows developers to create an efficient business application using either annotations or XML files. In annotations, the functionalities are used to map the java entities into relational tables, and the same operation is performed in XML through complex tags. Due to the flexibility of injecting code using annotations, Hibernate can override the existing mapping scenarios by mapping enum into the database columns [5] and also assign a single property in the java entity to multiple columns in the back-end (RDBMS).

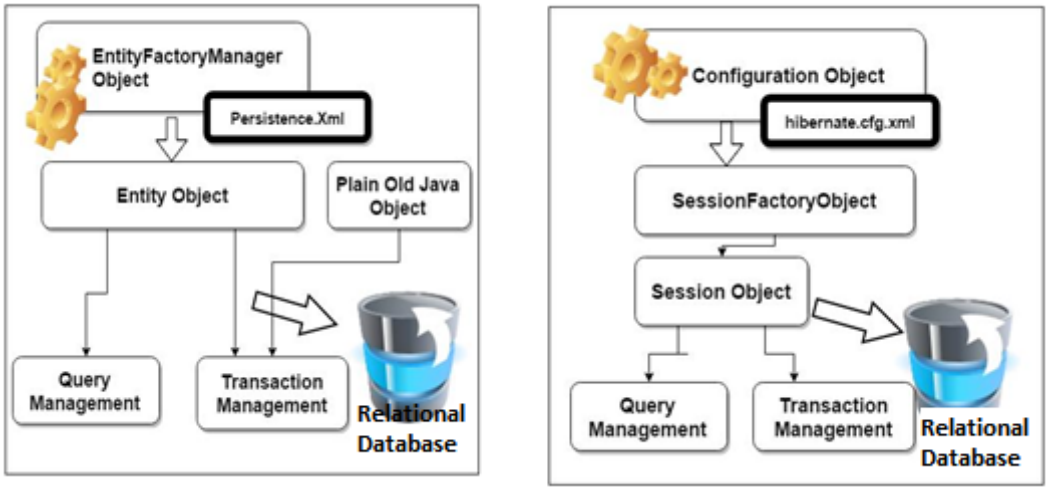

Figure 2.2: Hibernate JPA High level Architecture

However, this process needs a standardized approach, which is implemented in Hibernate JPA through standard interfaces and JPA programming. As a fully compliant platform with Technology Compatibility Kit (TCK) [5] to accomplish database modeling in JPA, Hibernate provides ORM mapping of the API with JPA classes and interfaces [13], and it has gained popularity as a platform independent portable language. With the Spring Framework, it provides a persistence layer for complex enterprise applications, and a template to encapsulate sessions into the template methodology [13]. J. E. Lascano in [3], found Hibernate could use Persistence.XML configurations to initialize the properties of a JPA project with the org.hibernate.ejb.Hibernate Persistence provider.

A typical Java Standard Edition platform (Java SE) persists databases through JPA using the EntityManagerFactory object, a static method to initialize the persistence process. This object is similar to the SessionFactoryobject in Hibernate API and is used to invoke the java entities for persistence performing the basic CRUD operations by using the EntityFactoryManager instance object. Figure 2.2 describes a Hibernate JPA high-level architecture with ORM mapping using the Hibernate API with a session, and an EntityManager Object using JPA. Subsequently, Hibernate configures a connection using the EntityManagerFactory or SessionManagerFactory object. The transaction and query management objects in Hibernate can perform run-time data retrieval operations to execute queries. According to research on JPA 2.1 [14], Hibernate JPA facilitates functionality to a native API through advanced object-oriented programming. These OOPS concepts enhance the functionality of conventional API by handling inheritance and polymorphism through implicit superclass mapping platforms and execute complex

SQL queries using Java Persistence Query Language (JPQL).

2.2.2 Design and Model of EclipseLink JPA Implementation

EclipseLink JPA, or Eclipse Persistence Service project [7], is a vendor independent performance oriented ORM solution through advance plug-ins and features. EclipseLink JPA provider began as an Oracle's TopLink product and was adopted by the java community as an API that enables developers to build efficient EclipseLink JPA APIs. In [19], D. Clarke states that EclipseLink currently provides ORM mapping solutions with JPA by binding Object-XML Moxy (with support for JAXB) [19] and SDO (Service Data Objects) [19] methods and interfaces, using org.eclipse.persistence.annotations package [19]. A study by T. Giunipero [21] found that EclipseLink JPA easily shares the same domain model with multiple persistence, providing metadata for multiple services. Also, the EclipseLink JPA default manages JAR, eliminating the compatibility issues in explicit downloads. As a comprehensive JPA ORM solution, EclipseLink is a strong standard with several advanced features and database extensions, including stored procedures, Native SQL Queries and data types [7]. A study of JPA implementation by L. Sembera [18] found that EclipseLink's advanced database extensions through SQL compliant features, such as triggers, views, and indexes, improved the efficiency of the API. Apart from existing features, EclipseLink JPA supports advanced functionality, including coordinated shared cache, clustered databases on RDBMS and non-relational databases [31]. As well as existing JPA interface support, EclipseLink weaving techniques simplify the manipulation process by using bytecode [31] processed at run-time or compile time. Weaving not only adds an API process to help the java entities move data to and from the database, but it also configures the Java class enhancements to identify lazy loads, and changes in the fetch data groups to perform internal optimization [31]. By default, EclipseLink supports run-time enhancement by allowing entities to efficiently enforce primary key/foreign key relationships, and manage to join two or more entities. The mapping process in EclipseLink is categorized as follows:

Project class: A holder used to persist mapping and configure metadata.

Descriptor: Holds mapping information for each data member that EclipseLink should persist or transform to.

Map file: Includes a project.XML file, and is associated with sessions that EclipseLink can use at run-time for byte-code enhancement.

As a standard JPA implementation, EclipseLink uses the EntityManagerFactory object

to create connections, and the EntityManager object to map java entities to a relational database for CRUD operations. Figure 2.3 describes an EclipseLink JPA high-level architecture. The persistence in EclipseLink configures a connection with the Entity-Manager object, which is then invoked by the EntityManagerFactory object. Configuring a connection in EclipseLink is straightforward since the objects of Transaction and Query interfaces are used to retrieve and execute complex SQL operations based on the EntityManager object. EclipseLink JPA includes additional capabilities like weaving to persist data into the database [30]. This technique manages schema creation and maps java classes into the relational databases to increase EclipseLink JPA API efficiency.

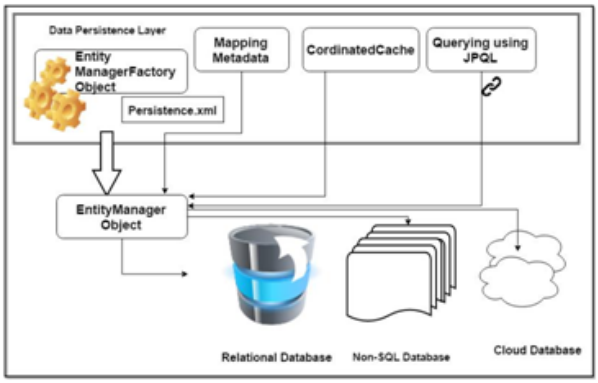

Figure 2.3: EclipseLink JPA High level Architecture

2.2.3 Design and Model of OpenJPA Implementation

OpenJPA is an open source, lightweight JPA implementation [6] integrated with an Apache server [6], and an OpenJPA API automates mapping procedures and schema creation by using SynchronizeMapping in the persistence.XML file [6]. OpenJPA documentation in [6] states generally that a sequence generator allows users to increment the java class through seq interface as a primary key column. The generator interfaces include additional capabilities to store time in the primary key, with the TimeseededSeq [6] and random hex string using the UUIDHexSeq [6]. [6] also showed that, by default, OpenJPA domain models are inefficient for handling the entity constraints. However, OpenJPA re-constructs constraints with Schema Factory [24], which provides the functionality to persist data. According to OpenJPA designers, a standard JPA implementation depends on monitoring java entities, but the specification does not define how to implement them. Some OpenJPA providers auto-generate innovative sub-classes or proxy objects [24] on the entity objects at run-time to improve the monitoring

process, and others use byte-code weaving technologies to enhance the actual entity class objects automatically. OpenJPA manages run-time enhancements by creating sub-classes which work with the small API, but they degrade performance and cause functionality defects in a real-time production environment. Another OpenJPA problem was described by M. Enoki et al. [10], who proposed that caching in OpenJPA is a coarse-grained level that results in low cache hit rates. An improved caching strategy was introduced to adjust indexes with of fixed size at the granular level, which improved the performance of the cache by referencing objects dynamically [10]. This mechanism in cache invalidation improved the performance with adjustable indexes for openJPA APIs accessing data, using small index size with available cache memory. Although Caching in OpenJPA lacks generality and has two levels: data cache that retrieves the entities loaded from the databases, and query cache that stores the primary key column returned in the transaction for reference [6]. Dividing the cache into levels has increased cache coordination between java entities, and when the frequency of database search queries is high, caching manages CRUD operations before using the content of the database. The second level caching adds a layer in OpenJPA to allow an overview of a cache object, which eliminates the need to explicitly handle developer's requests. According to [6], an OpenJPA implementation persists java classes using either annotations with property declarations, including @column, @id and @version [6], or through XML files. The @column is the column name, @id is the unique identifier similar to the primary key, and @version annotation is common practice to ensure data integrity during merges and also acts as a confident concurrency control. The persistence process also prompts the EntityFactoryManager and EntityManager objects to complete the persistence process. The EntityFactoryManager object is a JPA standard implementation to create a one-time connection, while EntityManager wraps dynamic transactions for persistence. Figure 2.4 shows OpenJPA high-level architecture achieved by defining the persistence process, using the JPA standard specifications through the EntityManger and the EntityFactoryManager Object. The figure incorporates two-level cache strategies in OpenJPA (Query and Data caches), to optimize API efficiency.

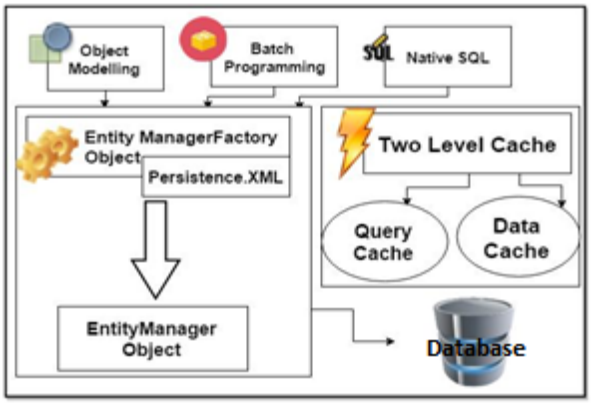

Figure 2.4: Open JPA High level Architecture

2.2.4 Design and Model of DataNucleus JPA Implementation

DataNucleus JPA is an open source java persistent framework that the saves the state [20] of a java object in the relational database. The DataNucleus JPA is considered a fully compliant JPA implementation [20] providing transparent data persistence and byte-code enhancement to persist data in the database. Through transparent data persistence, DataNucleus JPA directly manipulates data in the relational database using the java entities. The persistence process is divided into two stages: First phase enhancement applies the common technique of byte-code manipulation to create persisted java classes [20]. DataNucleus manages the enhancement using the DataNucleus enhancer, which manipulates the java classes by creating metadata [25]. In the second phase, the persistence process performs a schema formulation using the DataNucleus Schema tool that generates a metadata file and uses annotations to persist the java entities. The tool accomplishes persistence through the "datanucleus.autocreateschema" property [25], which is defined in the persistence.XML file. DataNucleus schema generation depends on the enhancer to create metadata for every class and sub-class in the API, and the metadata file contains information about the persisting units of the API. This increases the efficiency of the mapping process and allows developers to perform abstract processes by simple method calling. Though this programming style is useful in distributed environments, it is complicated in a centralized architecture [20]. A major distinction between DataNucleus and other JPA implementations is DataNucleus support for schema creation property. It provides flexibility to treated schema manually through the command prompt, or by java programming using annotations or XML files. The process is highly sensitive and allows do not allow duplicate jar files. The compatibility issues in

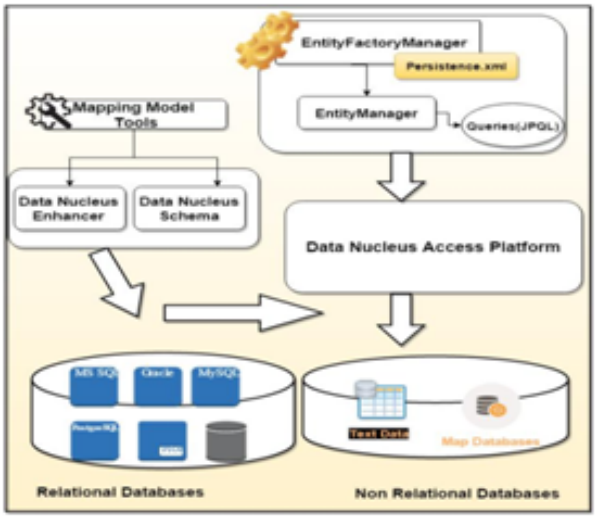

Figure 2.5: DataNucleus JPA High level Architecture

DataNucleus can restrict metadata creation, so to avoid problems and manage schema developers usually configure the enhancements manually. According to DataNucleus, their API must include the following jar files to complete the persistence process: datanucleus-core-2.1.1.jar, data nucleus-enhancer-2.1.0-release.jar, data nucleus-JPA 2.1.0-release.jar*, datanucleusrdbms2.1.1.jar and asm-3.1.jar * [25]. According to DataNucleus developers, the persistence processes in DataNucleus JPA is incremental [20], which means not all fields are retrieved immediately. The process has evolved into steps to improve the efficiency of the overall application using the entity graph [20] and lazy loading. The graphs are further managed by standard JPA interfaces called Entity-ManagerFactory objects and EntityManger. Similar to other JPA implementation SQL operations, such as CRUD operations in Data Nucleus is managed by the EntityManager. Figure 2.5 shows a high-level DataNucleus JPA architecture with EntityManager objects performing persistence. A typical DataNucleus JPA implementation persists java classes via metadata management [20], which is the second step defined using traditional XML files or annotations. XML files are the preferred mode of object mapping because it involves less deployment time to retrieve and store data in the database. DataNucleus doesn't define a mechanism to cache a query, but entity graphs in DataNucleus Cache data object through generic compilation [20]. This is the initial step in the DataNucleus JPA caching strategy, because it is independent of the underlying data source, and creates expression trees. The next Java entity cache process creates a compiled expression tree, which is converted into the native language of the relational database. Finally, the

executed query returns the cached object with the results.

2.3 Comparison of ORM based JPA Implementations

2.3.1 Query and Transaction

A transaction is a sequence of operations executed by a unit of work performed in a relational database management system (RDMBS) [30], following basic RDBMS policies called the ACID property (i.e. atomicity, consistency, isolation, and durability) [30]. A JPA specification identifies every transaction as part of the mapping process and executes queries on the entities to retrieve data from the database. According to J. E. Lascano in [3], every transaction either retrieves or sends data to the database, regardless of the underlying data source. JPA uses the Java Persistence Query Language (JPQL) to query entities through the java object. To understand the complex features in JPA implementation, and compare transactional factors to familiarize developers with the functionality so they can perform SQL operations, Table 2.1 provides detailed analysis. The comparison indicates that OpenJPA provides a wide range of options for querying databases, but due to bug issues in the language other JPA implementations, such as Hibernate and EclipseLink, are preferred for optimized results. In [3], J. E. Lascano also states that JPA manages large data sets optimally by reducing the SQL code, thereby avoiding SQL injections by executing the code at runtime. However, joining two relations or performing sub-queries in a database is high cost, and requires proper construction and planning in the API. Every JPA implementation follows a built-in default fetching strategy, or custom database extraction strategies. These eliminate the need to build and manage fetching models, which can improve performance.

2.3.2 JPA Object class

Java Persistence API embodies collection packages with different interfaces, to create lists, sets, collections, maps, tree maps [23] and other calculative operations. According to OpenJPA developers in [6], lists and sets are the most widely used utilities to perform basic data retrieval tasks from the object model into the database model. Every JPA implementation supports a default option while retrieving information from the database. Hibernate, Eclipselink and DataNucleus accomplish data retrieval through set classes,

while OpenJPA operates on collections with performance overhead when retrieving data. In [19], D. Clarke states that lists without an index in Hibernate and EclipseLink JPA are handled as bags, which degrades API performance when loads increase. Table 2.2 compares all four JPA implementations based on lists and sets and identifies the API set to execute the SQL operations.

2.3.3 Mapping and Distributed Database units

In JPA objects, modeling is performed on objects called entities, which have relationships defined among them. Mapping an object forms an association between two or more entities, and each has a defined role in creating a relationship to the database. For entities, relationship cardinality defines the constraint specific to the number of relationships of JPA model java class mapping to the relational database, using sub-classes and super-classes [12]. Table 2.3 shows a comparison of all four JPA implementations, with various optimization features and techniques, including garbage collection, pagination, batch processing, auditing, and logging, to track relationships and manage the cardinality/ordinarily of the entity to improve the mapping process. Every implementation follows a rational approach, but since the relationship between the entities has more effect than the other factors, it is important to tune the entities before intricacy occurs. Hibernate support advanced features to accomplish OOP concept including database mapping as shown in Table 2.3, while other implementations like OpenJPA and DataNucleus JPA inherit complicated operations for simple table-per-class strategies [20].

2.3.4 Auto-Insert

Marking a field with the @Generated Value annotation limit sets value in the RDBMS to auto increment [5]. In a JPA implementation, defining a primary key to uniquely identify a row in a relation uses the auto, identity, sequence, and table values features. Every value specifies a behavioral pattern, and auto adds a global number generator [5] in the ID column for every java entity, as well as an identity incrementer that auto generates values with an exception. The entire process is automated to add a primary key column into the database so it is optimal and efficient. Table 2.4 on JPA auto-insert compares all four implementations based on a major database feature called a sequence incrementer, and compares the JPA implementations based on auto-generation in the ID column.

2.3.5 Connection and Configuration

Configuring a connection in JPA is accomplished by setting properties in the persistence.XML file, and using the EntityManager and EntityManagerFactory interface objects. According to J. E. Lascano [3], java persistence API grows well in any environment, regardless whether it is an in-house intranet for a few users, or developing a complex JPA API for thousands. JPA manages the connection with the EntityManagerFactory, which handles bottlenecks to open or close a connection and attach or detach the allocated resources. J. E. Lascano in [3] also stated that the connection pool executes the query in a data cache to implement persistence in the JPA providers. Every JPA implementation is assigned a set of connection protocols and fetching strategies, to optimize the resource utilization process. Table 2.5 compares connectivity and configuration protocols and default values in all four JPA Implementations, and determines which provides the most flexibility by customizing connections or setting properties, and includes information about the default values that are designated on a specific API. JPA implementation has a customized connection pool setup through advanced protocols such as C3P0 [5] and protocol [5], that make C3P0 a third party library for the JPA application. It can also be used to manage connection pools with other JPA implementations, such as OpenJPA, EclipseLink JPA and Hibernate JPA.

2.3.6 Performance Optimization

Performance Optimization in JPA implementation contributes some major criteria to develop an API, including an optimizer, fetch strategies, indexes and parameterized search options. To manage different implementations, JPA includes pre-loaded metadata repositories, which help perform tasks such as metadata mapping and improve query response time. Other feature that improves JPA performance is the JPA audit for tracking and logging information of every operation in the JPA API, for example with Hibernate interceptors and EventListeners [5] a developer to accomplish auditing. Table 2.6 compares JPA implementations based on performance optimization factors to develop robust applications. The comparative analysis includes features that are key to designing performance strategies when developing an optimized API. The features provide an opportunity to compare the most useful features a developer might need to improve query processing.

2.3.7 Cache

JPA caching processes enhance the performance of the API during the inflow and outflow of data, or when executing SQL operations on the database. J. E. Lascano [3] stated that caching in JPA is implemented through a series of updates, and developers need not consider refreshing the cache or any comparable activities related to caching or flushing. JPA can use distributed cache frameworks [3], such as ecache, memcache, cacheman or other JPA implementations, which means a developer does not have to allocate time for completing the cache services [7]. According to [7], JPA cache processing is divided into three levels. The first level (L1) cache is described in the persistence context for a session or EntityManger, and the second level (L2) is the coordinated or shared cache [36]. The third level is implemented in DataNucleus only, in the form of a result tree to generate a query in the JPA persistence layer. M. Enoki et al. [10] found that caching in JPA implementation is managed through a flushing technique that re-initializes the internal SQL cache, and executes the command creating cache invalidation problem. To improve the cache performance, strategies defined in [10] create adjustable indexes that can improve caching technique in JPA, though the problem still exists when the API update frequency is high. Hibernate query cache manages to cache the output of SQL queries through the primary keys. In EclipseLink, entity caching [19] requires third party assistance, while OpenJPA and DataNucleus query optimizers explicitly manage the cache when the query executes. Table 2.7 differentiates between all four JPA specifications, by comparing various caching features.

2.3.8 Threading

A multi-threaded standalone application persists data into a database and manages threading issues related to CRUD operations. Table 2.8 is a comparison of different JPA specifications to manage threads in a multi-database environment. Every JPA implementation has an EntityManager object to create queries and an EntityFactory-Manager to configure a connection to the database. However, a single instance of Entity Manager from multiple EntityFactoryManager threads is a wrong approach, because there is no guaranteed to be thread safe. Access to resource level transaction via EntityTransaction is bound to EntityManager instance. So the result is that we share the same EntityTransaction and use it to serially manage multiple transactions to start and end multiple transactions but with many threads. In Hibernate (4.1.4), references are stored to Entity Transaction instance field in the AbstractEntityManageImpl

Table 2.1: Query and Transaction in JPA

JPA Providers / Features	Hibernate	EclipseLink	OpenJPA	DataNucleus
Transactions optimization	Fetch optimization techniques and patterns with checkpoints	Change Tracking for transactions .	Aggregates and projections Of queries using count and Sum.	Native SQL Named Queries+ Stored Procedures
Use Fetching	Lazy by default, can be set to eager. EAGER: Convenient, but slow LAZY: More coding, but much more efficient	Lazy mapping Default but can make it eager.	Eager default fetch can be changed to Lazy. Strategy could be None, join , parallel	Fetch groups DataNucleus. "Default Fetch Group" : defined in all JPA specs, specifying the fetch setting for each field/property (LAZY/EAGER) and Named Fetch Groups : standard in JDO but a DataNucleus extension for JPA.
Query Parameters instead of encoding search data in filter Strings	Named parameters + other SQL options including Wildcard characters.	PERSISTENCE_UNIT _ DEFAULT (true by default)	With aggressive caching of query compilation data, and the effectiveness of this cache is diminished if multiple query filters are used where a single one could have sufficed.	Native queries are used for selecting objects from the datastore.
Large data set Handling	Hibernate Pagination Hibernate ScrollableResults Native SQL Each has its own advantage and disadvantage.	Pagination is one technique used to handle data sets.	By default, OpenJPA uses standard forward-only JDBC result sets, and completely instantiates the results of database queries on execution. Add on will only bring required objects.	Native SQL, JPQL, JDOQL provides this and the implementation in DataNucleus allows extensions for query handling in large data sets.
Query and Transaction Management	Manual Transaction Management. Can be automated with transactionManager. PROPAGATION_REQUIRED option can be set to avoid dirty check on data. Use HQL	Work in unit of work from a session with isolation level.To Query use executeQuery •Nested Unit of Work •Parallel Unit of Work	OpenJPA uses optimistic semantics by default, but supports both optimistic and datastore transactions. OpenJPA also offers advanced locking and versioning APIs for fine-grained control over database resource allocation and object versioning.	Work in unit with Local transactions, JTA transactions container managed Transactions and Spring managed transactions
Tune fetch groups	Uses lazy select fetching for collections and lazy proxy fetching for single-values.	Pre-defined fetch groups at the Entity+ Dynamic (use case) fetch groups at the query level	Load all data and leave large fields(binary, additional join)	fetching objects you have control over what gets fetched
Database indexes	Support cluster and non-cluster indexing	@Index annotation to index	Manual +Build-in	IndexMetaData+ optimized Index

Table 2.2 JPA object class

JPA Providers Features	Hibernate	EclipseLink	OpenJPA	DataNucleus
Use set instead of List/collections	Default used SET recommended by Hibernate creators, but List is a better option. Other options List, Set, Array, Map, bag and ibag—are all supported by Hibernate	Use set default but can add JPA class.	Default collection cause overhead, use Set, SortedSet, HashSet, or TreeSet , if duplication is not an issue	Set is default for collection of data . Can use any other List, Set, Array, Map .

Table 2.3 Mapping And Distributed Database unit in JPA

JPA Providers Features	Hibernate	EclipseLink	OpenJPA	Data Nucleus
Inheritance	Hibernate supports the three basic inheritance mapping strategies: -table per class hierarchy -table per subclass -table per concrete class. - concrete class strategy, - concrete class using Implicit polymorphism. Concrete Polymorphism doesn't support join fetch.	Type of inheritance: Single Table Inheritance, Joined Table, Table per Concrete	Mapping inheritance hierarchies to a single database table is faster for most operations than other strategies employing multiple tables. Strategy SINGLE_TABLE, JOINED, or TABLE_PER_CLASS.	The default JPA strategy is SINGLE_TABLE. Specifying identity of objects in the root of persistable class cannot be redefined again in the tree
Composite Persistence	JPA 2.0 likely to give Abstract Method error with Hibernate [4].	Composite Persistence unit for relational and non Relational database+ clustering.	Simple persistence.xml with connection parameters.	Allow Run time persistence using JPA using the one Data Nucleus with multiple persistence Unit. Also support JTA and non-JTA datasources.

Table 2.4 Auto-Insert in JPA

JPA Providers / Features	Hibernate	EclipseLink	OpenJPA	DataNucleus
Sequence Increment	Sequence style generator With increment more then 1 With following options IDENTITY SEQUENCE(best option not much restriction) TABLE (SEQUENCE)	Sequence number pre-allocation enables a batch of ids to be queried from the database simultaneously to avoid accessing the database for an id on every insert. Default Value: 50	Large bulk inserts Sequence overhead. own Sequence factory can further optimize sequence number retrieval.	+Need to Set Validate when cached property to false. +auto identity generator Recommended . + sequence default can be non-optimum set the,key_cache_size= 10

Table 2.5 Connection and Configuration in JPA

JPA Providers / Features	Hibernate	EclipseLink	OpenJPA	DataNucleus
Connection Pool	Build-in + third-party open source (c3p0, proxool)	Not Default ,Build-in (max/min 32 with initial 1 connection)	Need Third party+ Plug-in support	Not Default but third party data-Source DBCP,C3P0,Proxool,BoneCP,JNDI, and lookup a connection data source
Configure connection appropriately	Need to set up hibernate.cfg.xml file with properties.	Configure connection Property Add-on Connection Retry option	Default: on-demand	Three auto start options at classes, xml, schema and table .
Detached State Manager	Supports Transient, Persistent and Detached Objects	Explicit Detach, cascade Detach ,Bulk Detach.	Off By Default, pros needs enhanced persistent classes and the OpenJPA libraries at client tier	Data Nucleus fetch groups to control the specific fields to detach.
Always Close Resources	Session need explicit closing, Bean close automatically	Need to close explicitly or will create Memory Leak Problem	Garbage collection+ application level cleaning	Auto managed by Data Nucleus

Table 2.6 Performance Optimization Features in JPA

JPA Providers **Features**	**Hibernate**	**EclipseLink**	**OpenJPA**	**DataNucleus**
JVM optimization	Hibernate statistics with better GC.	Hotspot compilation modes and maximum memory.	Alter parameters in JVM like hotspot and compilation modes results in better performance.	Linked hashmap saves 4% CPU time. Hotspot is another option.
Preload MetaData Repository	Callbacks and Entity listeners methods are annotated by a callback annotation.	MetadataSource Adapter	By default, the MetaDataRepository is lazily loaded which means some locking. Put option load metadata upfront and remove locking.	Schema Auto-Generation at runtime or using Data Nucleus Schema Tool.
Enhancer	Byte-code enhancer(run and compile time) Maven, Ant , Gradle.	Weaving (Run +Compile time).	Build-time or deploy- time enhancement. Post-compilation Byte code enhancer	Default enhancement before runtime. Support Transparent Persistence - Run +Compile Time.
Enable Logging/Disable	Enable for performance analysis Log4j jdbc + jboss- logging (warn, error & fatal)	(eclipselink.logging .level) Values (Off, severe, warning , info, config , fine, finer, finest, all) . This optimization feature lets you tune how EclipseLink detects changes in an Entity. Default Value: AttributeLevel if using weaving (Java EE default), otherwise Deferred.	OpenJPA logging facilities, can enable SQL logging by adding `SQL =TRACE` to with `openjpa.Log` property. OpenJPA can optionally reformat the logged SQL to make it easier to read with `PrettyPrint=true` .	Log4J +set categories : Persistence Transaction Connection Query Cache, Metadata, Data-Source, schema native Schema-tool, JPA IDE Value Generation Recommended: DataNucleus category to OFF
Logging Performance Tracking/ Auditing	(Default) Envers for Tracking Previous version, Individual Entity.	Change Tracking Type: ATTRIBUTE, OBJECT or DEFERRED +Auditing ways: AUDIT_USER and AUDIT_TIMESTAMP column. Full history support	JDBC performance tracker(set to false)	MBeans internally to track changes via JMX at runtime Or It own API for monitoring

Table 2.7 Cache in JPA

JPA Providers / Features	Hibernate	EclipseLink	OpenJPA	DataNucleus
Flushing	Auto by default , Other mode Commit, manual, never, Always	Default (off) flush-clear.cache + Drop, DropInvalide, Merge	Automatically flush before queries involving dirty objects will ensure that this doesn't happen.	Flush.mode to AUTO (default) but allow manual handling of "n" objects
Data Cache(2 level)	Clustered cache, JVM-level (SessionFactory-level) cache on class-by-class and collection-by-collection, using any strategy read-write cache, Non strict-read-write, transactional cache	Cache with no locking, no And cache refresh. --session cache(default)+ query +cache(size, .invalidation) By default, + EclipseLink caches objects read from a data source.	Data and Query caching(optional cache). -Not related to the EntityManager cache - Data cache can operate in both single- and multi-JVM environments.	By default the Level 2 Cache is enabled + mode of operation of the L2 cache default UNSPECIFIED , others include ENABLE_SELECTIVE , DISABLE_SELECTIVE, ALL, NONE
Utilize the Entity Manager Cache	Default 1st level cache Add-on 2nd level and Query cache.	Default is not shared (Entity manager)+ shared object cache option in EclipseLink.	RetainState configuration option to true, using build in cache.	Need to explicitly close connection detachOnClose to set to True
Query Cache	Default Disabled, To Turn set Property to ,values=True	Option need to be Configured manually.	Default Disabled, To Turn set Property to ,values=True +supports Concurrent Query Cache.	Generic compilation includes a tree which is database independent.

Table 2.8 Threading in JPA

JPA Providers / Features	Hibernate	EclipseLink	OpenJPA	DataNucleus
Multi threading	Don't use hibernate managed objects in multiple threads. Use ID column	Handled but time consuming.	Single-thread default, can be set using openjpa.Multithreaded	Persistence Manager multithreaded. Default value is false.
XA(distributed transaction)	HibernateTransaction Manager	Time out problem.	XA slower than standard transaction, but support non-XA and XA transaction.	Not efficient in handling distributed transactions

class, but that is merely implementation detail [4]. EntityFactoryManager, however, is synchronized, meaning that an object is managed through the JPA API to allocate and de-allocate resources. It also includes a comparison of JPA implementations, based on distributed transaction (XA) [30]. A JPA API based features are defined in Table 2.8.

2.4 Conclusion

JPA implementations allow programmers to build extendable APIs by reducing and reusing the code to achieve data persistence. JPA also decreases the overhead of managing to map by bridging dissimilar modeling architectures [2]. High-level abstraction, however, is seldom too complex for an unfamiliar developer. The intent of this chapter was to consolidate the benefits and features of the four JPA implementations: Hibernate, EclipseLink JPA, OpenJPA and DataNucleus JPA.
A literature review about JPAs is advantageous for an unfamiliar developer, and existing JPA API developers can develop efficient and performance-oriented applications. However, the question of which implementation is best suited to which environment remains. This chapter provides a feature analysis of four JPA implementations, and concluded that Hibernate JPA is the most advance JPA implementation, with its helpful documentation, popularity and libraries to manage operations. DataNucleus JPA, on the other hand, lacks popularity among developers and researcher. Both the implementation provides a lot of opportunities to develop a strong performance-oriented JPA API. On the contrary, OpenJPA has open bug and error issues as reported by the OpenJPA website forums. However, performance discussed in the following chapters analyses which implementation is more dominant in JPA, based on complexity and performance.

Chapter 3

Performance Analysis of JPA Implementations (5 major SQL Queries)

This chapter reviews the performance analysis of different existing JPA implementations based on the ORM framework. Large enterprises working on JPA are concerned about the potential pressures and overloads that JPA implementations such as Hibernate JPA, EclipseLink JPA, OpenJPA and DataNucleus JPA can manage. JPA plays a significant role in the efficient performance of API since the persistence process can be coordinated with the advanced features used to sustain heavily loaded applications. Analysis can demonstrate the impacts of different queries in a JPA API through I/O, CPU, garbage collection and other determinants that affect JPA performance (e.g. threads (live/dead). Several JPA API companies, such as Hibernate JPA, EclipseLink JPA, OpenJPA and DataNucleus JPA, have recently been adopted as advanced JPA-based technologies by persistence providers, and many more will be expanding into the JPA market over the next few years. Though JPA has been around for many decades, it has experienced fluctuating popularity. The major advantages that have made Java persistence popular are vendor-independent programming and the amount of coding required to persist data into the database [3]. These factors have shifted the focus of developers to building APIs based on JPA, as the use of JPA coding has increased significantly the programming costs have been reduced. JPA has grown rapidly, and due to performance breakdowns, there is momentum for developers to conduct performance analyses to sustain JPA programming indefinitely in the near future. Furthermore, in today's software market there is increasing

pressure on developers and enterprises to adopt API to decrease development time and programming costs and to use more sustainable technology. There are a variety of JPA implementations, which have lead to many acronyms that are worth summarizing.
Currently, Hibernate JPA, OpenJPA and EclipseLink JPA are deemed the most successful performance oriented JPA implementations based on the Lazy Load concept, though DataNucleus JPA is involved in using metadata mapping through enhancers [20]. Hibernate JPA is typically the not most efficient of the four, but due to its advanced features and flexibility, it is considered the most scalable. Though DataNucleus JPA is the most advanced and prominent type of implementation, it is very similar to Hibernate and incorporates the same components. Unlike most Hibernate JPA APIs, DataNucleus JPA is efficient but it involves complex persistence processes. OpenJPA is solely an Apache server based implementation and lacks performance due to bug and patches issues [8]. EclipseLink JPA APIs, which are less common, are created in JPA libraries and stored as a part of the API. This study focuses on performance analysis of the JPAs that are implemented using JPA persistence interfaces. JPA offers numerous advantages over conventional Java APIs, including more efficient caching, lower programming costs, decreased garbage collection on entities executing the SQL query, lazy loading and less memory overhead required to support an API during peak times [3]. The main advantage of using JPA API is that it includes a lot of security and locking alternatives [43]. However, JPA and JPA implementations are improving rapidly, and with their advanced features, they are expected to become the technology that gains acceptance over other Java-based applications.
The performance analyses of JPA implementations will impact the configuration and operation of the applications. Here, we compare the efficiency of JPA implementations based on many intricate SQL queries (e.g. Parameter Passing, Pagination, Aggregations, Indexing). The results of this approach help evaluate the effects of a query with a particular implementation and highlight reliability weaknesses, which are useful for overall smart design practices.

3.1 Introduction to Performance Comparison

3.1.1 Hibernate JPA Performance Oriented Implementation

Hibernate is an emerging ORM framework that bridges the gap between two dissimilar architectures (i.e. Java classes and database tables). As the new standard for developing

Java programming with byte-code conversion, dynamic compiling at runtime, and two-level caching, Hibernate JPA provides high-performance code generation [5]. The advanced features output complex SQL queries with efficient JVM cycle, and with an advanced compiler, the native byte-code is easily converted into metadata. As a resource-oriented API language, Hibernate provides the flexibility and effectiveness to perform tasks with intense abstract levels and to load and persist classes and objects dynamically at run-time [5]. Other loading formats such as columnar selection the eager load which generates the schema at compile time affirm that Hibernate outperforms built-in threading support, and decreases code vulnerability due to session-objects-per-thread, or session in JPA [5]. Although with high-level abstraction using interfaces and classes Hibernate JPA increases encapsulation, it also limits the understanding of the developer at the class level. To optimize the Hibernate JPA implementation, a developer must consider the following:

- For performance in Hibernate JPA, we indexed some query with the optimal execution plan, using the advantage of processing queries in batches [32]. Another performance-oriented feature was pagination, which executes queries page-by-page.

- Hibernate provides the flexibility to use its own personal query language, known as Hibernate Query Language (HQL) [3], which further optimizes the code and performance. However, Java Persistence Query Language (JPQL) built on a JPA platform is more versatile in terms of libraries and repositories and executes every query using JPQL using ID generators for the sequence. We used ID generators with JPQL to obtain query results.

- JPA performs persistence using the EntityManager and the EntityManagerFactory objects through the annotations tags or XML files [32]. We used annotation which is neutral for JPA language and auto enhancement, which makes the JPA API more adaptable and suitable for any JPA standard environment.

- Hibernate is considered the optimal solution for mapping, as it offers a series of predetermined properties to fetch data in batches of specific sizes [32] while loading a process to execute queries and return the results back to the client API. We used One-to-One, Many-to-One, One-to-Many mapping strategies with lazy loading.

3.1.2 EclipseLink JPA Performance Oriented Implementation

A JPA implementation based on an EclipseLink or Eclipse Persistence Service project, as stated in [7], is an advanced provider that empowers Java developers to develop API solutions with optimized JPA binding methodologies, using Object-XML Moxy through JAXB [7]. Furthermore, service methods such as SDO (Service Data Objects) enhance the reliability of the application, by allowing the developer to use one language that provides the access to execute operations simultaneously on another implementation using the persistence framework. According to D. Clarke in [19], the EclipseLink delivery model provides multiple run-time environments for the API through the EntityManager class, with built-in libraries in Eclipse IDE decreasing the complexity of maintaining JARs and eliminating the adaptability concerns of an application. Due to the EclipseLink exceptional tools, the process of handling advanced database extension features, such as stored procedures, native SQL queries, and indexes, are easily implemented in the JPA API improving the performance [31]. The following optimization features were implemented in the EclipseLink JPA API:

- EclipseLink JPA supports advance functionality, including coordinated shared caches and two-level caching for databases using both SQL and non-SQL databases [7]. Developers often manage databases using coordinated shared caching, which provides consistent throughput of the distributed JPA APIs over multiple persistence units [7]. In the experiment, we used two-level caching to improve the persistence process.

- We used logging performance statistics tool to help log the summary of all the individual operation including the total time spent fetching rows from the database [7], the number of objects handled per second [7] and the number of objects involved in the query [7].

- Batch writing and caching pool are two major properties that execute operation in batches. Both the properties were set with correct value based on the load and query.

- We used compile time weaving to improve both JPA entities and Plain Old Java Object (POJO) classes for lazily loaded queries and complex fetch joins [7].

3.1.3 OpenJPA Performance Oriented Implementation

OpenJPA is a vendor independent, JPA implementation with Apache server [8], which provides developers the flexibility to automatically map SQL procedures and faster schema conception using a property called SynchronizeMapping [6]. Developing an OpenJPA implementation allows developers to auto-generate sequences through seq interfaces for primary key columns. The additional capabilities of storing time and primary keys using TimeseededSeq [6] and random hex string using the UUIDHexSeq [6], enables unique sequences and improves system consistency. The domain models of OpenJPA as stated in [24] do not maintain entity constraints, though the option of explicit constraint construction is possible with the Schema Factory property. A notable problem with OpenJPA is monitoring entities, which is resolved by using innovative, auto-generated sub-classes or proxy objects to enhance objects at run-time [24]. OpenJPA controls run-time enhancements in an API through sub-classes, which further degrades functionality and increases defects in the code. Following are some features implemented in the OpenJPA API to improve the performance:

- Caching in OpenJPA automatically flushes queries involving dirty objects, and guarantees coarse-grained caching with no locking or refreshing in a session cache to manage query and cache size for invalidation. An improved caching strategy that adjusts indexes with fixed sizes at the granular level was introduced in [10]. Though this creates an efficient caching process, by referencing objects at run-time improving caching with varying workloads. So we implemented indexes in some queries where it was necessary based on the query plan.

- OpenJPA implementation persists through annotation declarations with @column, @id and @version [6] properties. The add-on annotation was to indicate @column for the name of the columns, @id as a primary key and @version to ensure the integrity is maintained when merging and optimizing imitation and performing concurrency control. We used this option for reference while making changes to the original queries.

- To perform transaction optimization in OpenJPA, developer shave the option of aggregation or projections with the fetching strategy [6], for faster access using Eager or Lazy for accessing at run-time. We implemented both the strategies in the queries while designing the experiment based on the requirement.

- We use OpenJPA scrolling result sets [6] property to avoid using default forward-only strategy and to use the result sets strategy, which improved the performance without exhausting JVM memory [6].

3.1.4 DataNucleus JPA Performance Oriented Implementation

A DataNucleus JPA implementation saves the state of a typical Java object using an EntityManager object. Since DataNucleus is a fully compliant framework, it provides transparent data persistence [20] and byte-code enhancement to persist data into a database. It not only directly manages data cached in RDBMS, but also sub-divides the process into steps. First, the enhancement process uses byte-code manipulation to persist data, followed by an enhancer to control the meta-data [20]. The second step in the persistence process creates the schema using a schema tool, which automatically creates databases depending on the enhancer meta-data for every java class [20]. This process increases the efficiency of the JPA API by automatically mapping the java classes. The main issue was with the high sensitivity of DataNucleus JPA while handling redundancy in jars, or incompatibility among jars, we resolved the issue though an organized batch of JAR files, as well as asm jars to auto-manage the application. The following points were implemented in DataNucleus to optimize JPA API performance:

- DataNucleus JPA increments its metadata each time an item is requested by a client API through the run-time loading process known as entity graph [20]. The graphs standardize the JPA interfaces through EntityManagerFactory and the EntityManger object. Although the DataNucleus's mechanism to cache a query is implemented using entity graphs [20], applying a generic compilation of the data source creates an expression tree. The tree is compiled and converted into the user's native language, and the result of the executed query is returned to the object cached with a resultant [20]. The process was handled using lazy loading to fetch data along with fixed batch size.

- To manage cache Data Cache in DataNucleus JPA we used two levels (L1, L2). The two-level caching modes for operation provided a workable strategy which used 'enable selective' mode for improving the JPA API performance.

- We also managed transaction through JPQL, Native SQL and the advanced query method for fetching JPA fetch groups using stored procedure and indexes. With workable units, the transaction containers were initialized for query management

[20]. To tune fetch groups, indexes manage meta-data and optimized the access to the database object.

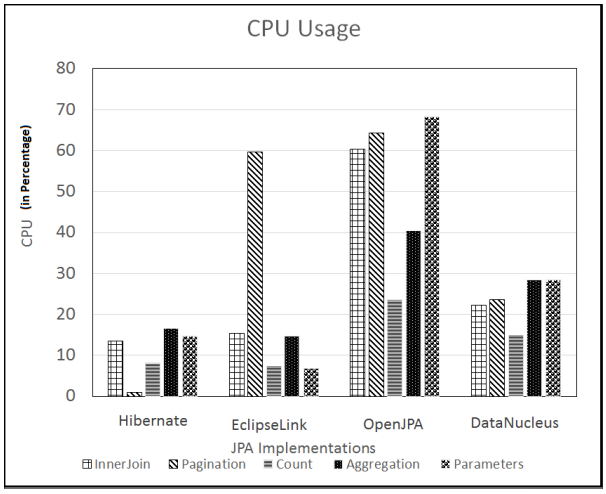

Figure 3.1: Metrics of CPU Usage in 4 JPA implementations performing 5 SQL complex Queries

3.2 Performance Analysis of JPA implementations Physical

The adoption of JPA implementation into a JPA API presents a number of challenges and potential reliability issues. Many practical questions have raised developers concerns, such as: Will the implementation be able to support the heavy load of a JPA implementation? And, what are the alternatives to costly SQL queries? Even without JPA, many software developers are frequently managing overloaded applications and adding advanced features to the situation which could avoid the collapse of the JPA API. JPA implementations can provide a solution to managing the persistence and performance problems, and this is currently under study. However, more detailed performance analysis of realistic SQL complex queries is required and is addressed in this chapter. With JPA implementation and the increase in the complexity of the API, performance could be significantly impacted, which would further affect the utilization of resources. Persisting data into the databases with excessive loads would determine which implementation has the highest likelihood of API failure. The proliferation of

JPAs into the Java programming could cause a number of undesirable side effects, including lack of adequate support for functionalities such as stored procedures and indexes, and a new querying strategy that could cause memory leaks. These impacts are studied in this chapter, through a performance comparison test based on 5 complex SQL queries. JPA implementations persist data into the databases and then retrieve it to execute SQL operations. Thus, to better understand the impacts of multiple JPA implementations through complicated SQL queries, extensive loads were used in all four JPA implementations to mimic a realistic user operation. The performance criteria applied to measure the APIs were CPU usage, I/O usage, garbage collection and Threads consumed by specific execution. This study had unique queries and performance parameters, which are integrated and analyzed simultaneously, the only focus being that a performance analysis would do comparisons and provide optimal results.

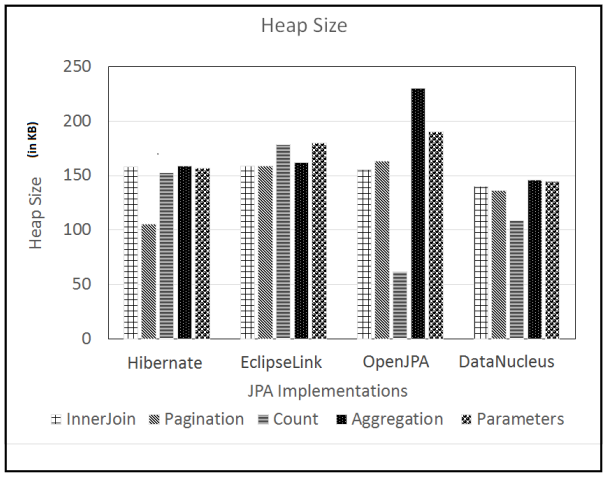

Figure 3.2: Metrics of Memory Size in 4 JPA implementations performing 5 SQL complex Queries

3.2.1 CPU and Memory Usage

In order to accurately evaluate the impacts and stresses of a query on JPA API, a realistic console based application (CUI) was developed. Unlike existing analyses, the impacts on the query, based on CPU and I/O, were evaluated to include the number of CPU cycles required for a particular query to execute (Figure 3.1). A CPU is the brain of any Java Virtual Machine (JVM) based application. A higher rate of CPU usage by a Java-based application means the API is degrading performance. With a CPU using Intel Core i7-7920 processor, a RAM size of more than 40 GB on a Microsoft Windows 7

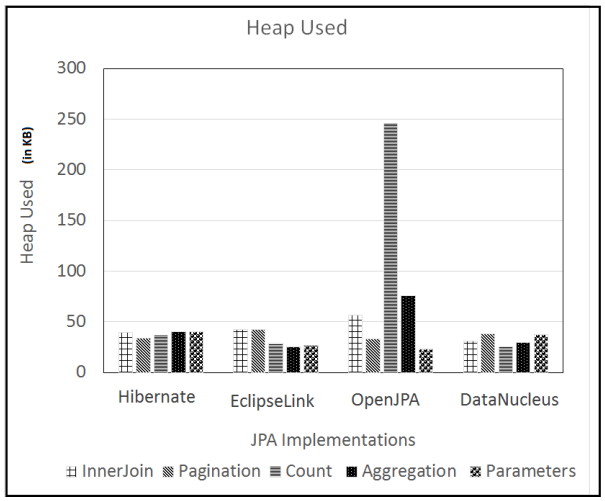

Figure 3.3: Metrics of Memory Used in 4 JPA implementations performing 5 SQL complex Queries

platform and Sun Java JDK 6 and above using 8 MB cache minimum the CPU cycle and memory consumption analysis in JPA APIs was averaged in the range of 0-80 percent. A significant number of CPU cycles is needed to execute a query, and demonstrate the output on a console. A developer can monitor and manage query performance of any Java-based application using Visual VM. To simplify the process, Visual VM monitors the same process and displays the output. For each implementation, the percentage of CPU usage verifies the optimal performance of the JPA API. Of the five queries, OpenJPAs consumed CPU below 40%, while queries like aggregation, pagination, inner join and parameter passing affected the performance with usage rates above 40%. Hibernate JPA and DataNucleus JPA showed smaller variations in query performance, with Hibernate JPA achieving results between 1% and 20%, and DataNucleus JPA between 20% to 30%. For the remaining implementations, EclipseLink JPA represented lumped loads from 4% to 15%, including an increase in CPU cycle while executing pagination query to 60%. The total percentage of all the five queries were averaged, and are presented in Figure 3.1.

Hibernate JPA and DataNucleus JPA outperformed the other implementations throughout the evaluation phase. Memory in JVM consists of 3 major segments: the heap, the MetaHeap and miscellaneous. Heap memory stores Java objects at run-time and allocates the data area to execute startup processes. By default, every heap size varies according to the implementations and number of classes in the query. A programmer with knowledge of heap manages the heap space, since analyzing the heap dumps is critical.

When a Java program starts JVM it retrieves memory from the Operating System (OS), or when a new operator or object is allocated memory, all the processes are executed through heap [44]. In a JPA API, the initial size of the heap and the maximum units it acquired at the end of a query affect the performance of the application.
Figures 3.2 and 3.3 indicate the heap size and heap used by the entities of all JPA implementations for selected queries.

3.2.2 Threads Count (Live/Daemon Threads)

Typical JPA implementations increase the load in an API based on the number of threads (Live/Daemon Threads). Threads are lightweight processes which simultaneously perform several tasks in the background [27]. These operations are either synchronous or asynchronous. In any Java-based application, threads perform some of the major performance-oriented, inter-process communications [27]. In a JPA implementation, threads determine the fetching technique while executing a query. All four JPA implementation load processes were either in lazy or eager mode [27] (Figures 3.4 and 3.5), with a scale distribution from 0 to 12 threads in Visual VM. To make the loads realistic, the live and daemon threads were profiled. The performance analysis of thread count was selected because it is expected that a higher thread count would be more affordable, and would likely improve the performance initially by avoiding the thread deadlock problem and thread pool configuration issues. Although all the implementations performed similar, Hibernate JPA, OpenJPA, and DataNucleus JPA executed inner joins and aggregation with a thread count between 8.5 to 11 for live threads and 8.5 to 10 for Daemon Threads. Although Hibernate JPA and OpenJPA implementation thread count has significant improvement when impacting the JPA API . A potential case is shown in Figures 3.4 and 3.5 when executing an aggregation query. Results in Figure 3.4 and 3.5 indicate that the efficiency of JPA API can also be significantly impacted by the number of threads executing a query to retrieve a results from a database.

3.2.3 Number of Classes Loaded

Given the JPA implementation introduced in Chapter 2, it can be concluded that performance analysis of the Java API would improve the efficiency of the application and the queries executed to retrieve data from the database. The number of classes loaded is a complex process in JPA API, due to, shared and local classes over a dynamic metadata. Analyzing the performance in terms of the number of classes loaded would

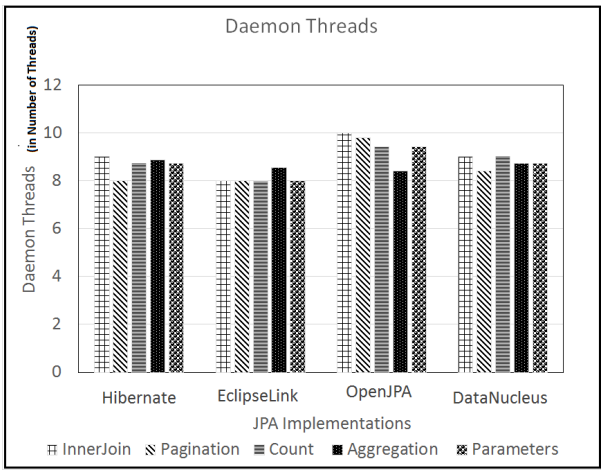

Figure 3.4: Metrics of Daemon Thread in 4 JPA implementations performing 5 SQL complex Queries

provide a developer with significant knowledge of JPA internal architecture. Visual VM presents criteria to measure the number of classes loaded when executing a specific query and determines the total number of objects allotted by class (including array classes) in a relation.

The total number of classes is loaded and defined by both the percentage of bytes, and the number of bytes allocated to each class. In a JPA-based API, entities are significant to the application. For each entity loaded to execute an SQL operation in the JVM, the profiler reports the size and number of objects designated since the profiling session started. The outputs are automatically refreshed as new objects are earmarked and new classes are loaded. For any Java application, it is important to understand the number of classes that are loaded for a particular query. Visual VM provides a graphical display of the information, which is then averaged (Figure 3.6). It provides a visual representation of all the loaded classes and the number of shared classes in a single application, on a scale of 0 to 6,000 classes. Although Hibernate used the maximum number of classes to perform operations, it still had optimal performance compared to other JPA implementations. OpenJPA used a smaller number of classes while performing a complex SQL operation (3,500 to 4,500 bytes), but it had too many trivial problems. Although DataNucleus JPA outperforms all the JPA implementations (between 3,200 to 3,600 loaded classes on average), it is not adopted as an API due to its complexity.

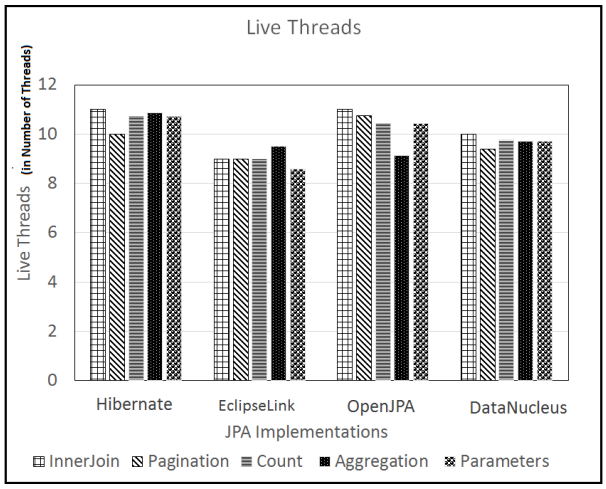

Figure 3.5: Metrics of Live Thread in 4 JPA implementations performing 5 SQL complex Queries

3.2.4 Garbage collection

In JPA, garbage collection is a process by which a JVM eliminates the dead objects from the Java heap space, and gives memory back to the heap; it is automatically implemented in all JPA frameworks [32]. GC in all four implementations exhibited distinctive results that corresponded to the complexity of SQL queries such as inner join, aggregation, association, pagination, and count. In JPA, the JVM is a distributed heap that is divided into three segments and sorted by the characteristics of the permanent data, the old data, and the live/active data [44]. New objects generate a heap allocation in the active domain. During the execution of a query, objects are either alive or dead. Objects that remain alive after the operation goes to a permanent location, whereas dead objects go to a garbage collector and are re-initialized. The permanent space in the JVM stores meta-data about classes and methods that have been persisted. Figure 3.7 depicts the average mean of five SQL queries performed on four different JPA implementations. Hibernate JPA performed well with less than zero percent garbage collection in some queries, while Open JPA and DataNucleus JPA lacked performance due to heavy memory leaks, particularly with respect to pagination and count queries.

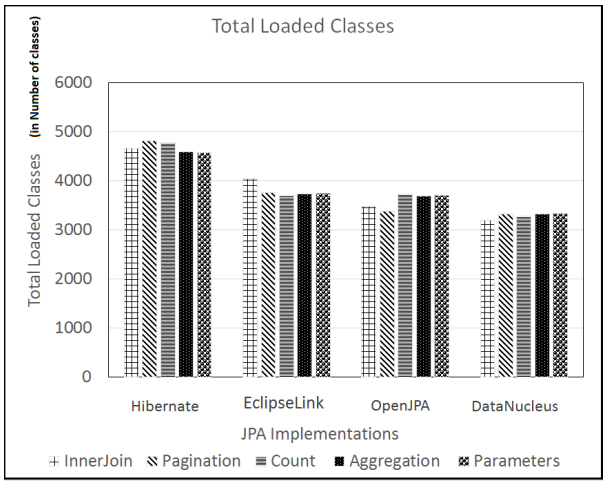

Figure 3.6: Metrics of Number of classes Loaded in 4 JPA implementations performing 5 SQL complex Queries

3.3 Conclusion

The goal of this chapter was to analyze the potential performance of existing JPA implementations for measuring the performance of complex SQL operations. The impacts of the JPA API, were based on performance parameters and load variations, using five different SQL queries, Eclipse IDE for entities and Oracle 11 g to persist data into a relational database. The main conclusions of this chapter are as follows:

- Hibernate JPA and EclipseLink JPA have significant positive impacts on performance in terms of CPU and memory usage, managing threads and garbage collection, even while executing the most complicated query in the peak demand.
- OpenJPA performance is most highly impacted when executing queries, and compatibility issues add further constraints with the Java 1.6+ version.
- Although DataNucleus JPA has shown significant signs of performance, its JPA implementation is not popular due to the heavy complexity of performing the persistence process. JPA interfaces and implementations are dependent on one another, and this provides new opportunities for developers to persist the Java application with minimal code. With the development of JPA, using Java applications as persistence processes will become more frequent and applicable for an API.
- From the perspective of a developer, this performance analysis will provide reliable

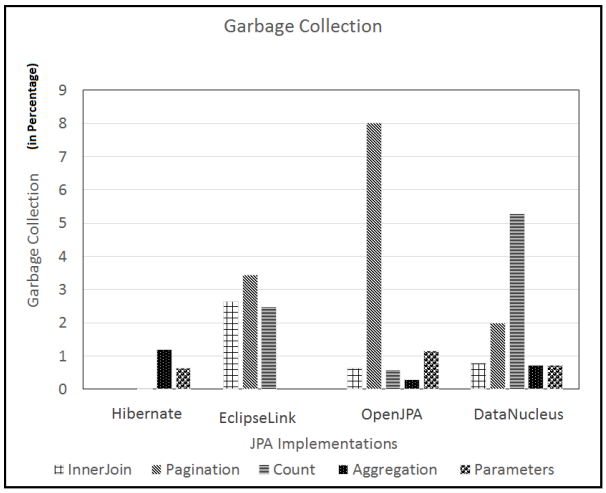

Figure 3.7: Metrics of Garbage Collection in 4 JPA implementations performing 5 SQL complex Queries

results based on JPA utilities, and lead to higher user satisfaction. Due to the existing challenges in software development and unwanted loads, development of low-cost software is essential to analyze the technologies and define preferences through properly detailed documentation.

In terms of optimizing Query Language in DataNucleus JPA, the implementation requires more control of its own transaction management system. OpenJPA requires more flexibility to handle complex queries and data types, and compatibility support for Java 1.7+ version. The results with EclipseLink JPA were difficult to retrieve, and the implementations require more stability due to the high crash rate in visual VM.

Chapter 4

Review and Analysis of JPA Implementation on Cloud

In Chapter 3, a performance analysis of existing JPA implementations on the local database was based on Oracle 11 g. Chapter 4 discusses cloud persisting platforms, such as DBaaS, for JPA implementation. Figure 4.1 shows a high-level architecture of DBaaS with three major components: Eclipse IDE with a JPA provider, Oracle Database 12c using SQL developer and DBaaS using Oracle 12 c cloud provider.

With the rapid development and expansion of information technology (IT) infrastructure in the recent decade, implementing and managing an application programming interface (API) has become a resource intensive task that transforms much of the software stack while developing and monitoring databases. Developers building a JPA API often adopt persistence platforms. These are either distributed or parallel, depending on the computing paradigm, capability, and operational cost. Generally, Cloud-as-a-service provides virtualization of physical resources in a logical environment, with the flexibility of scaling the database [36]. The aim of visualization through databases-as-a-service (DBaaS) was to improve application performance, reduce operational costs and provide reliable JPA APIs that can be accessed over the internet. For a JPA implementation that can operate on a local physical machine, partitioning one physical environment into several logical environments increases the usability of the database in remote locations, in terms of workload and storage. This not only allows work to run on several nodes or virtual machines (VMs) but also consolidates on one physical Node (Database).

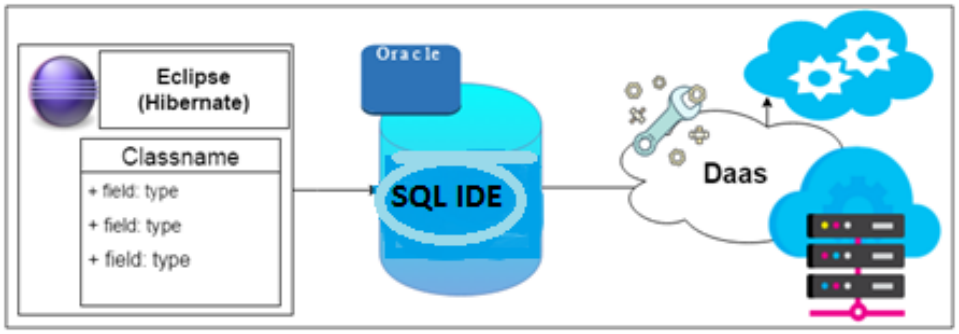

Figure 4.1: High Level Architecture of Database-as-a-service (DBaaS) for cloud database

The process virtually migrates a database service dynamically from one physical machine to another via the internet [36]. Typical cloud computing architecture is divided into three service level agreements: Database as a service (DBaaS), Infrastructure and Platform as a service (PaaS). These pay-as-you-go cloud computing services reduce the burden of storing and configuring a database management system (DBMS). For example, Amazon Aurora is an Oracle-compatible, enterprise-class database that stores data in a cloud at 1/10th the cost of commercial databases [40]. The Aurora database maintains up to 15 low-latency read replicas, 64 TB of automatic data scaling, high storage capacity and multi-directional replication across all zones [40]. Over the past few years, migrating to a cloud services platform has proven to be a powerful technique to achieve goals such as dynamic workload, balancing nodes, and facilitating management and patience over the internet [35]. Figure 4.2 shows the high-level architecture of Cloud-as-a-service. The diagram highlights User Interfaces(UI) accepting client requests. A Security Access Layer (SSH) creates a public-private key pair to create a communication path to a cloud and security [34]. The third layer is the Cloud administrator, is similar to the Database administrator in a physical database on a local system. In a cloud environment, Cloud administration handles some of the major tasks of assigning resources required by the application.

While resources are distributed by Cloud administration, a Load balancer creates fair balance and adequate distribution based on consumption and queries [34]. The three vertical layers on the right in Figure 4.2 define the database parameters for configuring and managing the distribution of resources effectively and creating roles, profiles, and users. To increase cloud service security, every user is assigned a different role and profile [34]. The load balancer layer handles heavily loaded requests from the client application and reduces latency without affecting performance. After substituting the load and

security, the final goal of a cloud service is to set the service level, which is the platform, infrastructure, and software that is implemented through a service level agreement based on a user's account.
The backup, compute resources and third party service (also on the right in the figure) provide a migrating developer with an advance tool to manage a database on the cloud. This chapter is a comprehensive study of migrating to DBaaS (Private/Public cloud) in a JPA implementation based on ORM framework, highlighting the features through benefits, cost and underlying characteristics with a Table 4.1 and Table 4.2 on two major cloud providers (Amazon versus Oracle). The intent is to understand the effects the Hibernate JPA, EclipseLink JPA, DataNucleus JPA and OpenJPA have using DBaaS, and to foster JPA API future development on clouds with a comparison with the local database.

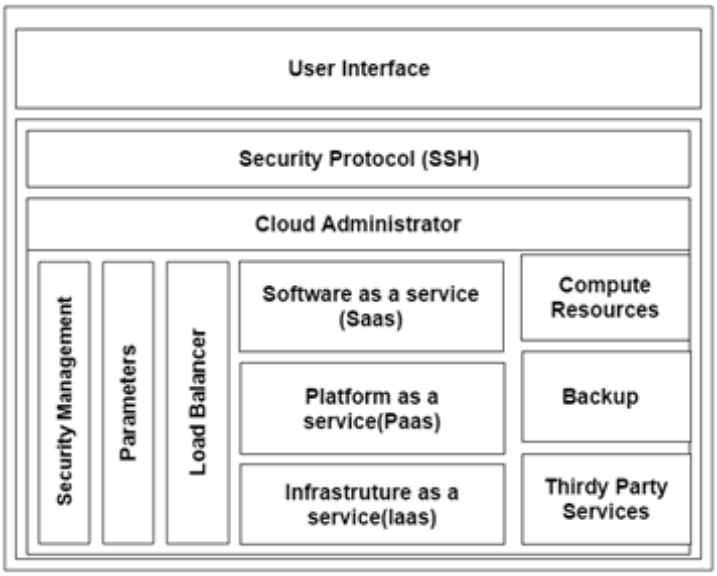

Figure 4.2: Cloud Computing Architecture Inspired by Oracle 12 Cloud Services

4.1 Analysis of JPA Framework

4.1.1 Hibernate JPA Local versus Cloud Database

- Central Processing Unit (CPU): A methods request involves a response time, which shows long it takes a query to execute, in milliseconds. The CPU time used for a query is a measure of the number of CPU cycles needed to compile bytecode into

Table 4.1: Feature Comparison of DBaaS (Oracle Versus Amazon) Part 1

Providers Components	Oracle Data Cloud Services	Amazon Cloud Services
Vertical/ Horizontal Scaling	Oracle RAC Clustering provides horizontal process scaling as service demands increase.	Allows Scale up/down but system goes down for few minutes (in maintenance).
Standardized Programmable Interface	Enterprise Manager, Cloud Control and DBaas interface	Amazon RDS API, or the AWS Management Console.
Grid Computing	Build on Grid computing architecture, server virtualization and dynamic provisioning	No option stating Grid computing
Number of Pools	Pool for database, schema and pluggable database	N/A
Security Management	SSH Tunneling (Public / Private key) using putty generator, port opening, SQL*Net, web access.	SSL , Type II (SAS 70 Type 2), X.509 certificate, SSL Protected API, Access control list.
Dynamic Structure	Resources elasticity + Pluggable Database Support all the features same as Oracle Database.	Dynamically increase or decrease accounting to the size of database. Goes offline for few minutes. Warning: -On a MySQL DB instance, avoid tables in your database growing too large. - On a MySQL DB instance, do not create more than 10,000 tables using Provisioned IOPS or 1000 tables using standard storage.
Capacity Management	Initial size 7.5GB RAM increase dynamically increase or decrease according to need.	Resizable capacity with initial 10GB with 5 GB to 6 TB of associated storage capacity.

Table 4.2: Feature Comparison of DBaaS (Oracle Versus Amazon) Part 2

Providers Components	Oracle Data Cloud Services	Amazon Cloud Services
Configuration Parameters	Username, Password Public IP address, Edition, version, Release and Parameter options.	Username, Password and Endpoints, Parameter Option. Configure IP address range, create subnets, and configurer outing and access control lists
Cloud Provisioning	Rapid and dynamic providing	Amazon RDS Provisioned IOPS (SSD) Storage
Data Compliance	Minimal administrative and configuration compliance.	On request, a report is generated which is recorded in the NDA documents. Some compliance support are- ISO 27001 and PCI DSS
Encryption	Data is encrypted by default in your tablespaces.	Amazon RDS manages the Oracle Wallet and Master Encryption Key for the DB Instance
Database Supported	Oracle 9i-12c	MySQL, MariaDB, Postgre SQL, Oracle, and Microsoft SQL Server DB engines

native code by the JVM. CPU time is an event parameter of the caller method and includes the CPU time of called methods and query execution. Garbage collection (GC) is another parameter and is measured by the average mean of time-stamps required for garbage collection by a query. Figure 4.3 shows the number of CPU and GC cycles in bytes, which are calculated with queries response times in the next section. The graph in the figure indicates local versus cloud databases and shows that the cloud database performed quite well in every query. The blue bars indicate that, on average, the cloud database took three times less as a number of CPU cycles to complete an operation than the local database. In terms of CPU cycles, a similar pattern was observed in Q1, Q2, Q5, Q6, Q7, Q8, Q9 and Q10 with local and cloud CPU cycle. The GC collection showed almost opposite results to the cloud database in Q1, Q2, Q3 and Q4 performing well. Overall, the response time of the local database was optimal as shown in next section along with CPU and GC cycle. The CPU and GC cycle was measured with a scale between (0 - 8) which was normalized using the percentage of usage between the range of 0 and 100 percent.

- Memory (Heap Used and Heap Size): Measuring the performance in terms of memory heap and MetaSpace in a Hibernate JPA implementation was complicated, due to heavy fluctuation in the memory graph. Figure 4.4 shows the monitored heap used for executing queries, to measure and trace query behavior. By default, the size of the heap depends on the type of loading, but a typical application tested using maximum Heap usage of 4849 MB with a MetaSpace size of 1602 MB. The overloads were terrible and reached the peak in DataNucleus JPA, although Hibernate JPA performance when executing queries in the cloud-as-a-service Q3 (Right Outer Join) was optimal, and heap used on a cloud showed efficient results for almost every query. The results proved that the cloud database was optimal in the performance and reducing overhead, whether the query is Left, Right or Full Join. Other implementation, however, were facing compatibility issues such as OpenJPA and EclipseLink JPA, while DataNucleus JPA was not performing well because of heavy memory leaks.

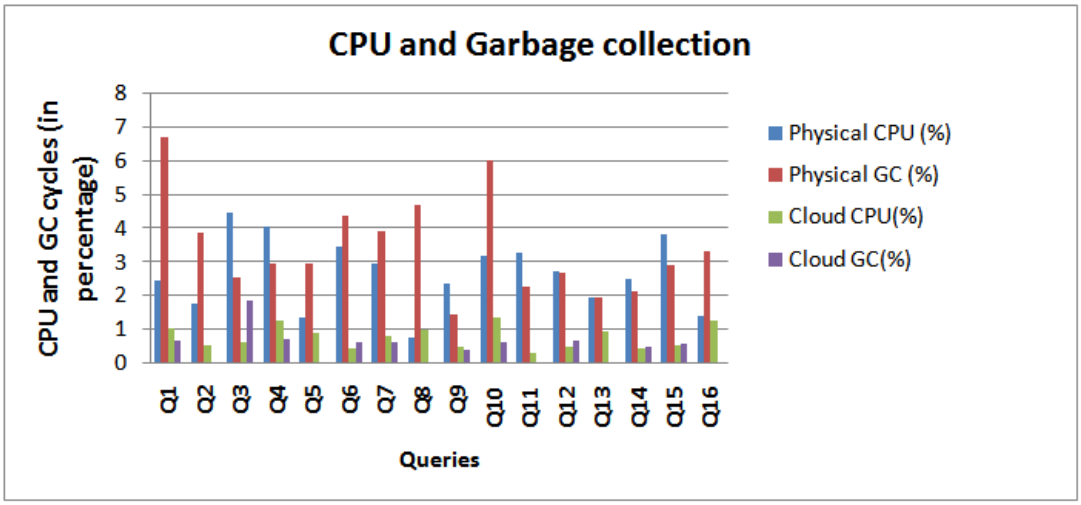

Figure 4.3: Number of CPU and GC cycle in Hibernate JPA using local versus cloud database

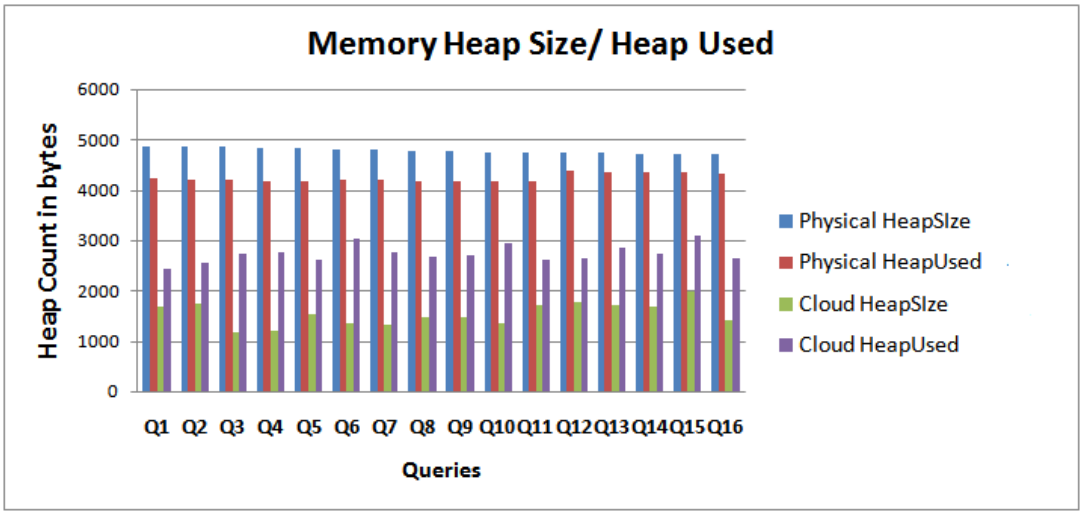

Figure 4.4: Heap used and Heap Size in Hibernate JPA using local versus cloud database

The extended results for Hibernate JPA is shown in Appendix D verifying parameter such as Number of classes and Thread (Live/Daemon) counts using local versus cloud database through Figure D.1 and D.2.

4.1.2 EclipseLink JPA Local versus Cloud Database

- CPU and Garbage Collection: The results of using the parameters as CPU and GC in local databases to measure the number of cycles showed more promising locally than those from the cloud database. The number of CPU cycles in Q1, Q2, Q6, Q7, Q11, and Q12 clearly showed that physical CPU performed well, and the cloud database usage was almost 3.8 times than the local CPU utilization. A similar result was observed in GC with query Q2, Q6, Q11 and Q16, where the local database outperformed the cloud database. Some of the 16 queries didn't execute due to compatibility issues such as lack of support. EclipseLink JPA does not support FULL outer JOIN, otherwise, the results with the local database would have been more understandable. Figure 4.5 shows the CPU and garbage collection for the physical and cloud databases.

- Heap Size and Heap Used (Memory consumption): The heap size is in bytes and ranges from 0 to 3500. Figure 4.6 clearly shows that the heap size in local and cloud databases is similar, with a very feasible difference of 500 MB. However, the amount of heap used shows that the local database consumed more metaspace than

the cloud database, which is due to the platform where the database is persisted (location and zone of the database). Although the difference between the local database heap size and the cloud database is not overly high. In Q1 for example, the heap size in the local database was near 1200 MB, compared to about 900 MB on the cloud database. Overall, the cloud database performed well with the heap sizes in query Q2, Q6, Q7, Q11 and Q12, and not as well in Q15, which was a complex Join operation with a heavy load. Hence, the results showed that heap used was optimally utilized on cloud database compared to heap size which showed the opposite.

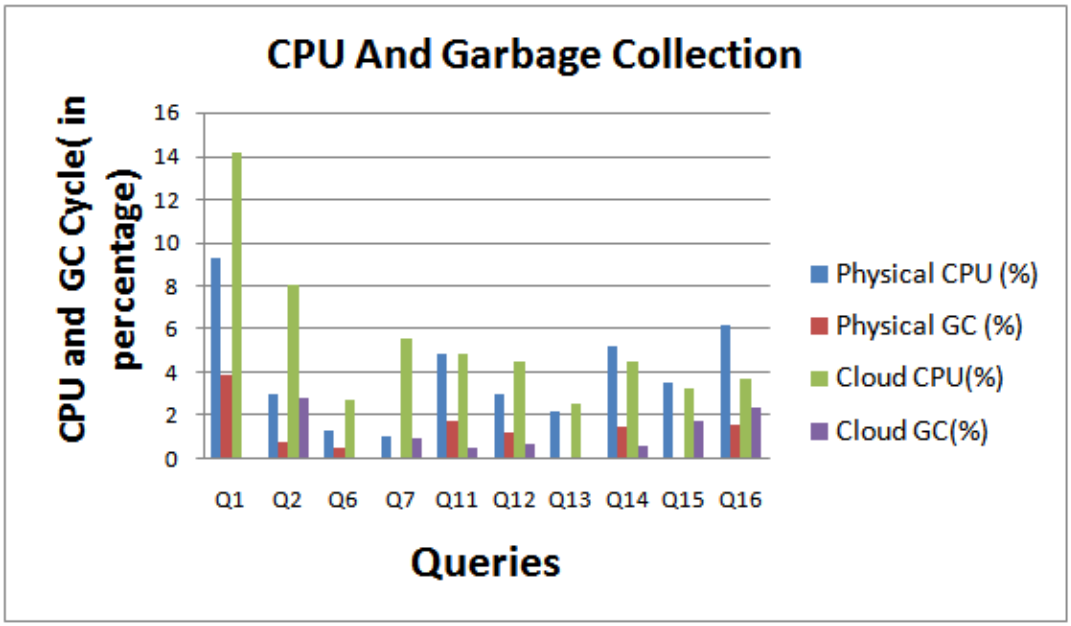

Figure 4.5: CPU and GC cycle in EclipseLink JPA using local versus cloud database

The extended results for EclipseLink JPA is shown in Appendix D verifying parameter such as Number of classes and Thread (Live/Daemon) counts using local versus cloud database through Figure D.3 and D.4.

4.1.3 OpenJPA Physical Local versus Cloud database

- CPU and Garbage collection: The results shown while analyzing the CPU consumption in OpenJPA implementation was very surprising, as there was a significant difference in the number of CPU cycles consumed by the local and cloud databases.

The performance of the cloud database was better, while the local database had high CPU cycles of upto 10.6 times in simple JOIN and OUTER JOINS queries. The GC collection results, however, were better with the local database. The severely affected local database performance in Q1, Q2, Q5, Q8, Q15 and Q17 (an extended query with an alteration to Q16 so the query could execute) showed worse performance in the local database compared to the cloud database, with CPU cycles as low as 1.9 times. This section shows that the local database executed the operation faster than the cloud database, which is surprising because features such as second level caching or batch size did not improve the CPU cycle as expected. OpenJPA has many compatibility issues, including Java version support, OutofMemory exception and a lack of query operator support. Figure 4.7 shows the CPU and GC comparison between the local and cloud database.

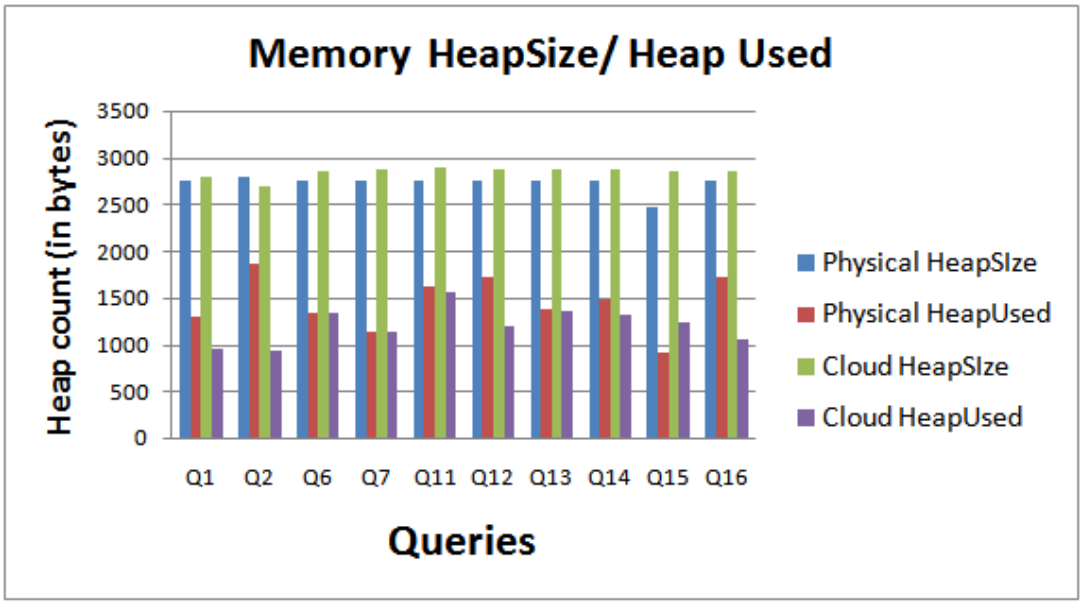

Figure 4.6: Heap used and Heap size in EclipseLink JPA using local versus cloud database

- Memory (Heap Size and Heap Used): Figure 4.8 shows a comparison between memory heap used and size to execute SQL operations on local versus cloud databases. The performance of local and cloud database is almost the same in all queries except Q8, where the local database has a heap size of more than 25,000 bytes, which leads to inefficient memory utilization and the program becoming non-responsive several times. The performance of the queries was quite low in local databases compared to cloud database for Heap used, while the heap size was

consistently good showing moderate fluctuations. Although OpenJPA gave many unexpected errors while executing operations using high response time in cloud databases as shown in Section 4.2. The performance of local database with Q1, Q2, Q3 and Q12 showed the same number of bytes for Heap size on cloud and local database, while the Heap used showed very less but some moderation for both the persisting platforms.

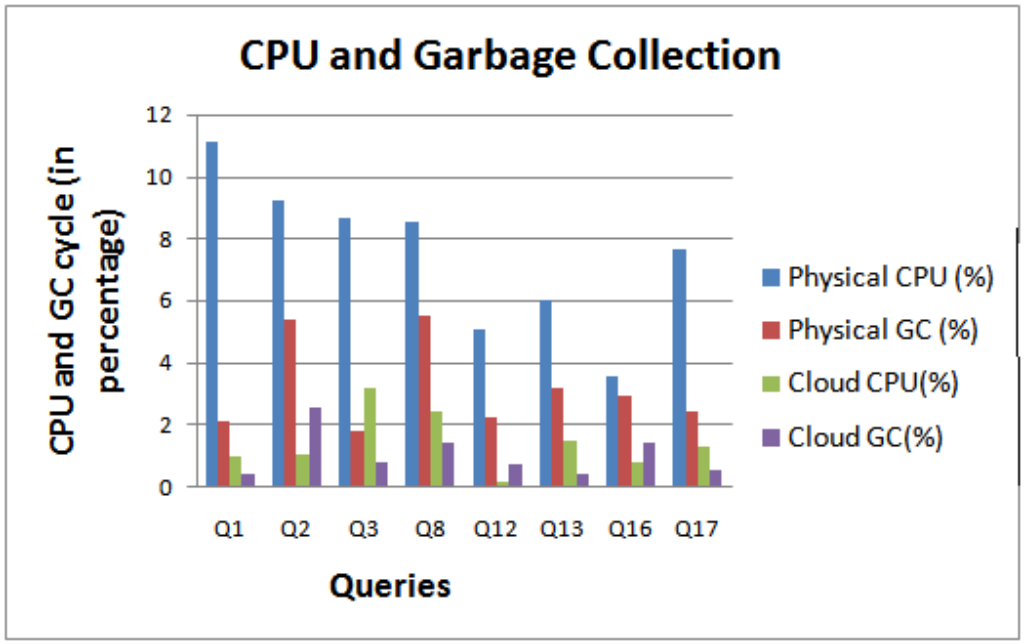

Figure 4.7: CPU and GC cycle in OpenJPA using local versus cloud database

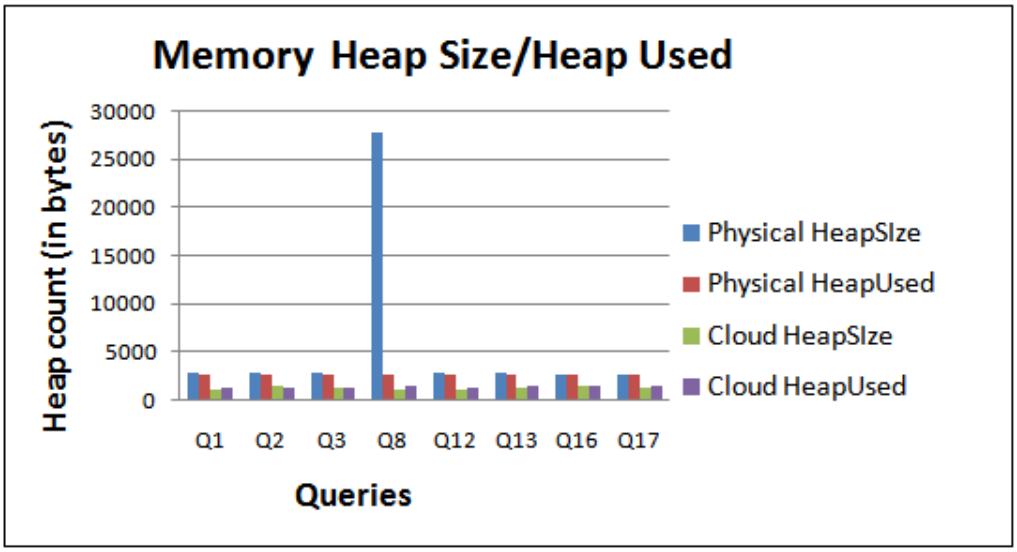

Figure 4.8: Heap used and Heap Size in OpenJPA using local versus cloud database

The extended results for OpenJPA is shown in Appendix D verifying parameter such as Number of classes and Thread (Live/Daemon) counts using local versus cloud database through Figure D.5 and D.6.

4.1.4 DataNucleus JPA Local versus Cloud database

- CPU and Garbage Collection (GC) : In DataNucleus JPA, where simple java classes are enhanced by creating metadata before schema creation, heavy CPU consumption is highly likely. Although right and full outer joins are not supported in DataNucleus, we observed that executing an SQL query in DataNucleus JPA with a local or cloud database wasted many CPU cycles when processing simple CRUD operations. However, in our experiment inserting more than 4800,000 records into a cloud database was time-consuming and enhancer did not add to an improvement in performance. Figure 4.9 shows CPU and GC of a DataNucleus JPA implementation persisting data using local and cloud databases. Some of the operation failed due to lack of support to perform complex union and difference. The graph shows that overall local database performed well compared to the cloud. The performance in Q1 and Q6 was considerably higher with the local database but the cloud had better GC collection and good response time; similarly, local database GC collection was moderate especially for query Q2 and Q9. The CPU percentage was on an average of 17% higher in Q9, compared to less than 5% with

the local database. Q8, Q9 and Q11 were worst in terms of CPU cycle consumption with the cloud database, while Q17, which was an altered Query 15 for DataNucleus JPA, had similar performance in local and cloud databases, with less than 5% CPU and less than half consumption of cycles with that for GC collection.

- Memory (Heap Used and Heap Size): Memory consumption in DataNucleus JPA performed well in the local database with heavy memory loads in and out of memory. We explicitly had to increase the memory size of Eclipse IDE, using the command prompt to execute queries in DataNucleus JPA on the local database. Figure 4.10 shows the heap size and heap used by DataNucleus JPA on local and cloud databases. In the graph, Q6 and Q14 illustrate the lack of performance of heap size and heap used is in DataNucleus on the local database while Q2 and Q8 performed well on cloud database. The DataNucleus JPA two-step method, using enhancer and schema creation, further increased the response time, and often hung the system and Eclipse IDE. Although Q1, Q11, and Q13 showed similar heap size, the performance of the DataNucleus JPA implementation with the cloud database was poor. The performance with the cloud database took almost ten times as long in the execution process, and we had to close others window operations and increase the memory size of Eclipse IDE to execute operations.

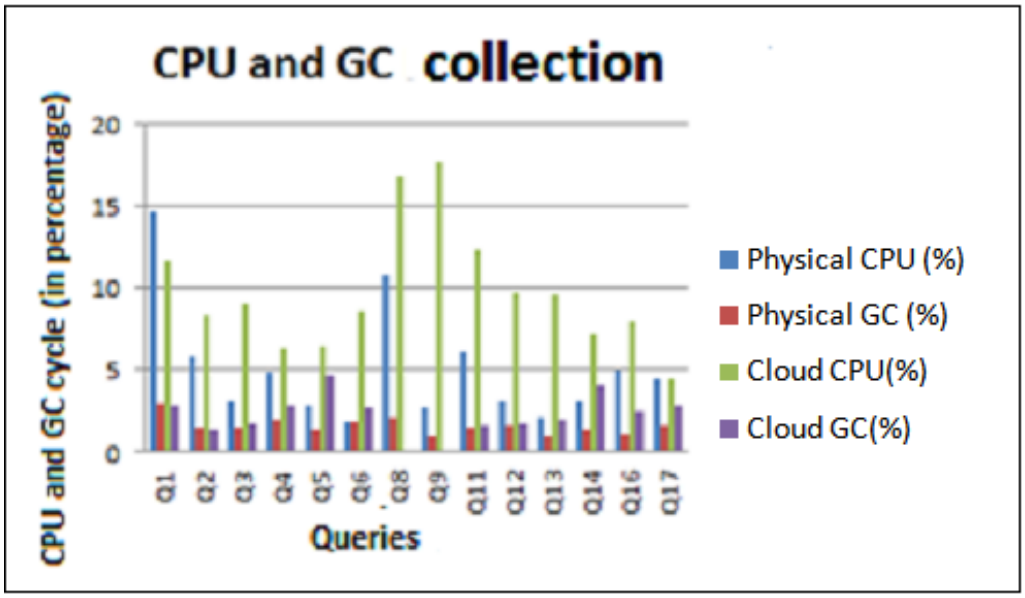

Figure 4.9: CPU and GC cycle in DataNucleus JPA using local versus cloud database

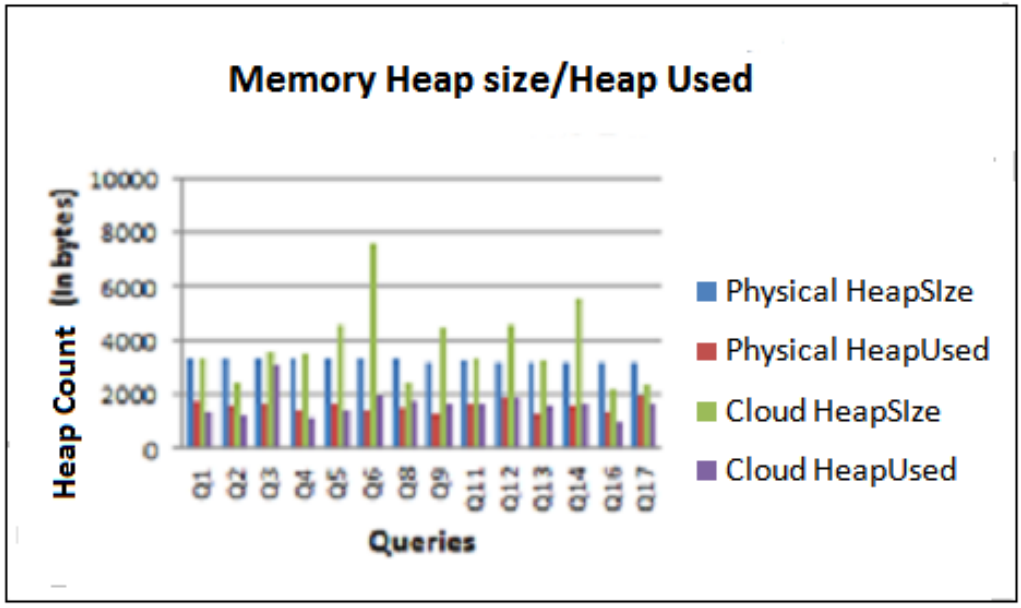

Figure 4.10: Heap used and Heap size in DataNucleus JPA using local versus cloud database

The extended results for DataNucleus JPA is shown in Appendix D verifying parameter such as Number of classes and Thread (Live/Daemon) counts using local versus cloud database through Figure D.7 and D.8.

4.2 Experiment Results

To test the performance of local and cloud databases using existing JPA implementations, query response times were measured with Data Definition Language (DDL) under different conditions. Each statement went through at least ten iterations, and for each, the query response times were noted and the average calculated. All the SQL queries were executed using the EMPLOYEE database, in Oracle 12 c and Oracle Cloud services. Running the SELECT statement retrieved data, and the number of results fetched in each case was tabulated along with the response time values. The CPU, GC, Number of classes loaded and Threads helped acquire the results through graphs. The entry was obtained by dividing all the entry response times with the initial entry (number of row) response time for each calculation. The graphs above indicate the number of cycles, the number of classes loaded and the number of threads needed to execute these operations below.

- Query 1: (SIMPLE SELECT)
  SELECT E.ID, E.Date, E.Salary from Employee E where E.Date = '11/11/2014' and E.ID>0 and E.ID <X
  The above query retrieves Employee ID, Employee Date and Employee Salary columns for the date 11/11/2014, with an Employee ID range of 0 and X from the Employee table.
  For convenience, the value is given as X in the query and the value of X is tabulated in Table 4.3. Table 4.4 illustrates the average elapsed query time for both a local database (Oracle 12 c) and a cloud database (Oracle Database-as-a-service) in Hibernate JPA.
  As the table shows, there is a significant change in the cloud and local databases in Hibernate JPA performance while retrieving rows from tables. These results indicate that the local database is performing well for this query. At 30,000 entries (rows), the response time of the cloud is almost double, and at 60,000 both databases show increased response times. At 120,000 entries (rows), the cloud response time is about double, and at 240,000 entries (rows) it is almost 1.5 times higher. At 480,000 entries (rows), the cloud database is again about double.
  These results for EclipseLink JPA show that the local database performs well for this query. At 30,000 entries the response time is twice with the cloud, and at 60,000 it is almost double. At 120,000 entries (rows) the cloud has a 2.5 times higher response time, at 240,000 it has three times higher response time and at 480,000 entries (rows) the cloud database is 2.5 times higher.
  The results for OpenJPA show that the local database again performed better. At 30,000 entries (rows) the response time in the cloud database is triple, and at 60,000 it is almost 2.5 times higher. At 120,000 entries (rows) the cloud has a higher response time of almost triple, and at 240,000 entries (rows) it has almost 3.2 times higher response time. At 480,000 entries (rows) the cloud response time is almost 3.1 times higher.
  The results in DataNucleus JPA show that the local database outperformed the cloud database for this query. At 30,000 and 60,000 entries (rows), cloud response time is about five times higher. At 120,000 the response time is almost 5.1 times higher, and at 240,000 and 480,000 entries (rows), the cloud response time is about four times higher.

- Query 2 (SELECT COMMAND USING SIMPLE JOIN)

SELECT E.ID,E.salary,E.firstName,E.lastName FROM Employee E JOIN Department D ON E.ID=D.ID WHERE E. Name = 'b%' and D.ID>0 and D.ID<X.

The above query retrieves data EmployeeID, Employee Name and Employee salary columns for the Employee Name = 'b% within Department ID range 0 to X , based on the join in Employee and Department table. The task of the above query is to pull out large number of matched rows in both tables, and retrieve data from the Department table between 0 and X. The average response time in the cloud and local databases is shown in Table 4.6, and the value of X is in Table 4.5.

These results show that the local database using Hibernate JPA is performing well for this query. At 30,000 entries (rows) the response time is five times more in cloud, and at 60,000 it is 5.9 times higher. At 120,000 entries (rows) cloud has a 6.9 times higher response time, and at 240,000 it is 6.5 times higher. At 480,000 entries (rows) the cloud database is 6.7 times higher.

These findings show that the cloud database performs like the local database in OpenJPA. At 30,000 entries (rows) the response time is about 1.4 times higher in cloud, and at 60,000 it is almost twice. At 120,000 entries (rows) the cloud response time is 3.2 times higher, at 240,000 it is 11 times higher, and at 480,000 entries (rows) the cloud response time is almost 4.6 times higher.

A similar trend in cloud and local database performance was observed in EclipseLink JPA. At 30,000 entries (rows) the response time is about 1.7 times higher in cloud, at 60,000 it is almost 1.9 times higher, and at 120,000 the cloud has an almost 2.5 times higher response time. At 240,000 entries (rows) the cloud response time is 2.2 times higher, and at 480,000 it is almost 2.5 times higher.

The DataNucleus JPA results show that the local database also performed well for this query. At 30,000 entries (rows) the response time is 3.3 times higher in cloud, at 60,000 it is 5.6 times higher, and at 120,000 it has 10.1 times higher response time. At 240,000 entries (rows) the Cloud has a 12.1 times higher response time, and at 480,000 the cloud response time is 10.9 times higher.

- Query 3(SELECT COMMAND USING INNER JOIN)

  SELECT E.ID,E.salary,E.firstName,e.lastName, E.Salary, E.Date FROM Employee E INNER JOIN EmployeeProjects EP ON E. PID =EP.PID and E.Name= 'a%' and EP.PID> 0 and EP.PID<X

  The above query retrieves the Employee ID, Employee Name, Employee Salary

and Date within the Employee and Project table with Employee Name = 'a%' and with Project ID range 0 to X with value of X in Table 4.7. The task of above query is to pull out a large number of rows from the two tables. The average response time values in cloud database and local database are shown in Table 4.8. These results show that the local database is performing well in Hibernate JPA. At 30,000 entries (rows) the response time is triple in cloud, and at 60,000 it is 4.2 times higher. At 120,000 entries (rows) the cloud has 5.2 times higher response time, while at 240,000 it is 4.8 times higher. At 480,000 entries (rows) the cloud response time is 5.7 times higher.

The error in Table 4.8 is that EclipseLink JPA is OutOfMemory when the number of entries (rows) increases. At 30,000 entries the response time is 5.5 times higher than the local database.

For OpenJPA, the results show that the local database is performing well. At 30,000 entries (rows) the response time is almost five times higher in cloud, and at 60,000 it is almost eight times. At 120,000 entries (rows), the cloud has an almost 13.5 times higher response time, while at 240,000 it is 14.3 times higher. At 480,000 entries (rows) the cloud response time is 15.5 times higher.

Similar results indicate that the local database is performing well for this query in DataNucleus JPA. At 30,000 entries (rows) the response time is 5.4 times higher in cloud, and at 60,000 it is eight times higher. At 120,000 entries (rows) the cloud response time is 19 times higher, while at 240,000 entries (rows) the cloud responses are 11 times higher. At 480,000 entries (rows), the cloud response time is 10.9 times higher.

- Query 4 (SELECT COMMAND USING RIGHT OUTER JOIN)
  SELECT E.ID,E.salary,E.firstName,E.lastName, E.Salary, E.Date, D.branchName, FROM Employee E RIGHT OUTER JOIN Department D ON E.ID=D.ID where E.Salary>0 and E.Salary<X

  The above query retrieves the Employee ID, Employee Name, Salary, Date and Branch Name within the Employee and Department table, based on matching records and a salary range of '0' to X. The task of this query is to pull out large number of rows from the two tables. The average response time values in cloud and traditional databases are as shown in Table 4.10, and by value of X in Table 4.9.

  These results show that the local database is performing well in Hibernate JPA.

At 30,000 entries (rows) the response time is four times higher with cloud, and at 60,000 it is almost 3.2 times higher. At 120,000 entries (rows) the cloud has almost 4.5 times higher response times, while at 240,000 it is almost three times higher.

Table 4.3 Data Entries for Query 1

	30,000 entries	60,000 entries	120,000 entries	240,000 entries	480,000 entries
X	25000	45000	100000	230000	440000

Table 4.4 Response Time Values of Different Entries for Local and Cloud Databases Query 1

Response time	Retrieved results	Hibernate JPA		EclipseLink JPA		OpenJPA		DataNucleus	
		Local Database (ms)	Cloud Database (ms)	Local Database (ms)	Cloud Database (ms)	Local Database (ms)	Cloud Database (ms)	Local Database (ms)	Cloud Database (ms)
30,000 entries	20	6	13	16	29	18	52	12	65
60,000 entries	35	9	21	25	43	31	73	29	142
120,000 entries	43	17	47	34	72	52	144	46	250
240,000 entries	94	28	63	46	132	75	245	67	279
480,000 entries	145	62	132	124	311	154	314	89	333

Table 4.5 Data entries of the Query 2

	30,000 entries	60,000 entries	120,000 entries	240,000 entries	480,000 entries
X	5500	11000	22000	450000	105000

Table 4.6 Response Time Values of Different Entries for Local and Cloud Databases Query 2

Response time	Retrieved results	Hibernate JPA		EclipseLink JPA		OpenJPA		DataNucleus	
		Local Database (ms)	Cloud Database (ms)	Local Database (ms)	Cloud Database (ms)	Local Database (ms)	Cloud Database (ms)	Local Database (ms)	Cloud Database (ms)
30,000 entries	2,543	310	1,582	221	394	432	610	532	1,746
60,000 entries	4,522	421	2,134	553	1,054	632	1,256	724	3,664
120,000 entries	8,933	783	4,145	452	1,130	744	2,433	675	6,953
240,000 entries	16,596	902	9,563	533	1,178	941	10,662	855	10,233
480,000 entries	10,057	2,132	19242	1,443	3,653	3,135	14,424	3452	55,232

Table 4.7 Data Entries of Query 3

	30,000 entries	60,000 entries	120,000 entries	240,000 entries	480,000 entries
X	10000	15000	13000	45000	55000

Table 4.8 Response Time Values of Different Entries for Local and Cloud Database in Query 3

Response time	Retrieved results	Hibernate		EclipseLink		OpenJPA		DataNucleus	
		Local Database (ms)	Cloud Database (ms)	Local Database (ms)	Cloud Database (ms)	Local Database (ms)	Cloud Database (ms)	Local Database (ms)	Cloud Database (ms)
30,000 entries	321	385	1,373	212	1102	564	2,833	422	2,883
60,000 entries	563	422	1,922	Error	Error	755	6,044	534	4,274
120,000 entries	762	892	4,677	Error	Error	953	12,864	965	18,335
240,000 entries	1,322	1,730	7,893	Error	Error	2,455	35,133	1,964	21,604
480,000 entries	3,642	3,245	15323	Error	Error	3,218	49,817	3,533	38,503

Table 4.9 Data entries of the Query 4

	30,000 entries	60,000 entries	120,000 entries	240,000 entries	480,000 entries
X	5,000	11,000	21,000	23,0000	321,000

Table 4.10 Response Time Values of Different Entries for Local and Cloud Databases in Query 4

Response time	Retrieved results	Hibernate		EclipseLink		OpenJPA		DataNucleus	
		Local Database (ms)	Cloud Database (ms)	Local Database (ms)	Cloud Database (ms)	Local Database (ms)	Cloud Database (ms)	Local Database (ms)	Cloud Database (ms)
30,000 entries	353	230	920	Error	Error	Error	Error	430	1,260
60,000 entries	536	492	1,570	Error	Error	Error	Error	692	3,242
120,000entries	774	622	2,799	Error	Error	Error	Error	842	5,473
240,000entries	1,553	1,130	3,390	Error	Error	Error	Error	1,830	7,863
480,000entries	3,242	2,245	6,902	Error	Error	Error	Error	3,245	10,233

At 480,000 entries (rows) the cloud response time is about 3.1 times higher.
Similar response time differences were observed for DataNucleus JPA with the local database. At 30,000 entries (rows) the response time is three times that of cloud, and at 60,000 it is almost 4.6 times higher. At 120,000 entries (rows) the Cloud has almost 6.5 times higher response time, while at 240,000 it is almost 4.2 times higher. At 480,000 entries (rows) the Cloud response time is almost 3.1 times higher.
The results showed errors in OpenJPA and EclipseLink JPA in Table 4.8 indicate the query was not supported by the language, and it had compatibility issues with RIGHT OUTER JOIN or with the condition. The error reported in EclipseLink and OpenJPA are due to lack of support for SQL query.

The results for Query 5 to Query 16 is shown in Appendix C. The Appendix C includes a description of the Query and a comparison of response time between local and cloud database.

4.3 Conclusion

Here, we have suggested enhancing the existing work on JPA API in the cloud environment with a modification to the current physical local persisting platform, which is capable of saving resources but is limited to the local system. In addition, a cloud-based environment for JPA API would improve online monitoring and management, particularly for databases moderate size, as verified by this analysis. Experiments show that local persistence takes less response time than cloud database, and when the optimization features are implemented in the JPA API there is a significant difference in CPU and GC cycle between the local physical persistence and the cloud database especially as shown in Hibernate JPA where cloud database performs better while other implementations showed that cloud database is optimal.

Chapter 5

Statistical Review and Analysis of JPA based Implementation

In today's market, JPA implementations are one of the fastest-growing software applications. However, understanding the performance of a JPA implementation based on an ORM framework is challenging throughout the development and maintenance process. This is due to the dynamic nature of the application, which includes metadata handling and asynchronous multi-threading as explained in the challenges. A rigorous statistical analysis over a start-up and steady phase, using JPA benchmarks and highlighting the impact of factors such as JVM and GC, is an opportunity to increase knowledge of the underlying JPA API. Furthermore, rigorous analysis using Pearson's correlation coefficient localized the statistics over CRUD operations. The results of the experiment showed that JPA implementations follow the accuracy and proximity of task completion, with high confidence and low computational cost using the troubleshooting tool. Despite JPA's flexible features in [31] some challenges, it currently has not undergone detailed research and comprehensive analysis. Popular tools such as Visual VM and Java code Tomography (JCT) [45] provide limited but exclusive capabilities to analyze metrics effectively. This chapter presents a rigorous, semi-automated procedure using JCT and Visual VM for statistical analysis, while increasing understanding of JPA implementations. Using a combination of automated and manual JPA code, we determined the trace behavior during a specific start-up and steady phase. Our technique traces the abstract behavioral model of the implementation, and preserves ephemeral and persistent associations between a database and API. The analysis was based on the correlation between components using Pearson's correlation

coefficient. A high correlation (0.71 to 1.00) indicates the proportionality between the execution and processing time of the operations, and a corresponding JPA performs the calculations and manages the event. We implemented our approach to increase developer's understanding of CPU and GC collection.
As far as we know, we are the first to present a general technique for obtaining metrics by coupling JCT and Visual VM using low-level, operation-based execution in four JPA applications with a statistical correlation model. This chapter builds on our earlier work, where we proposed JPA feature analysis and performed performance evaluations through empirical experiments [31, 32]. It extends the existing approaches and evaluations of JPA implementations, as we propose a statistical evaluation using Pearson's coefficient with 95% confidence through experimentation. Overall, this work makes the key enhancement of a general technique for capturing and presenting the complex statistical analysis of JPA based applications.
To build a novel analysis from the results captured in the steady query-driven executions and a correlation analysis with varying levels of granularity, we performed a strict statistical analysis. The balance of this chapter is organized in sections: Section 5.1 briefly discusses the challenges of the JPA implementations, and Section 5.2 provides an overview of the Persistence.XML file in all four JPA implementations. Section 5.3 outlines the experiment design of our JPA API in detail, presents JPA benchmarks used in the experiment and performed an initial event execution with start-up and steady rates. Section 5.4 provides final analysis using JPA standard benchmarks and scaling factors (i.e GC and CPU) to improve the confidence in the results and examines a correlation among JPA Benchmarks using Pearson's coefficient, and Section 5.6 presents the conclusions of the entire analysis.

5.1 Challenges and motivation Towards Analysis

- Challenge 1: Dynamic Meta-data: Database persistence in JPA is the most popular technique in the current persistence process, though both JPA and JPA implementations are not as precise as they seem because of the complex annotation injection and XML tags. Applications modeling dynamic meta-data typically extend in two ways to define the data model: annotations and XML mapping files. For simplicity, annotations are injected in the code to manage the object-relational mapping paradigm over the XML mapping. Hence, XML mappings are no longer considered superior, and increased focus is now on building JPA API with

annotations, although the dynamic meta-data created by annotations needs extra efforts while persisting. The information generated by the annotated injections is quite complex and persisted in the JPA layer along with the business logic, which accesses the code redundantly using the persisted information, thereby creating mapping ambiguities and delays. For example, to capture the maximum length of an address field in an entity or a validation condition, JPA requires extra patch files and explicit get() and set() to perform persistence. In addition to the meta-data, other issues in mapping involve storing and accessing data. A typical JPA API includes Java Bean or plain Java classes, with a segment field (both getter and a setter methods) for every column corresponding to the table. A large data-set that automatically generates a reverse engineering concept through the auto generation of JPA code has a major problem: in a mature design a developer often encounters challenges with manually adjusted bean or mapping code for modeled data. A better approach would be to manage meta-data manually, and separate automated dynamic mapping without affecting the business logic layer. This allows division of the mapping process, and objects provide more elaborate results.

- Challenge 2: Eager vs Lazy Fetching: Despite significant progress in JPA, the problem of persistence tier remains. With the advanced loading techniques to implement a general approach, and a racing mechanism to fetch the induced entities from low-level concurrency constructs, creating JPQL or HQL queries after mapping to fetch entities that are associated with a proper relationship fetching technique for joins or added queries, further burdens the generated query tree structure. This is a double-edged performance issue; the feature delivers a faster results by automated SQL, but it also affects performance. A fetching strategy for retrieving correlated objects in an application often requires navigation through an O/R mapping metadata tool, or being overridden by a JPQL query. Building JPA implementations with advanced features requires specialized semantics that typically load entities, and localized EAGER/LAZY operations among relationships. This process is similar for all JPA implementations, such as Hibernate which retrieves the associated relation using SELECT Query or Service classes with @ManyToOne,@OneToOne, @OneToMany and @ManyToMany [4] mapping strategies deemed on LAZY loading. Our experiment handles inherent fetch strategies for data-needed mechanisms that show precise and practical scales for hundreds of thousands of lines of code. Thus, we built Trace files [4] and Visual

VM file XML files, with scarcity warnings and general meshes of the dynamic language. To measure the associations defined by a LAZY execution with join fetch, we used JPQL operator with fetch-join to manage the right join request from the required record, rather than retrieving the entire table [4].

- Challenge 3: Multi-Threading [Live and Daemon Thread]: To test the behavior of a multi-threaded application, developers often used test suites. For JPA implementation, the delivery of multiple databases in a single iteration is identical, meaning the tasks are performed in a pool of threads with multiple iterations. The main thread, which has many identical threads in the antecedent, forms a loop and waits for LIVE threads to join the end of the Daemon thread. A thread is often accessed in a heap while loading initial libraries, but with JPA asserting performance through a pool becomes a challenging task. The main difficulty is the connections between different entities and methods, including the EntityManager and EntityFactoryManager interfaces, which are responsible for execution and modification of the database model. To understand the performance of thread operations and breakdown tasks, a manual model with traditional stack trace was used through Visual VM tool. This allowed logging of every thread execution in the white box environment with ability, to be analyzed.

- Challenge 4: Two-level Caching: When implementing and re-factoring code, modifications are performed and triggered through lower-level caching events, which uses Query trees [20]. A JPA implementations often delegated caching with specialized tools that provide an integrated second level caching environment into the core of the frameworks. In DataNucleus JPA (i.e. Data Cache and Query Cache) [20] second level cache stores query trees to reduce the load in the JPA API, and manipulated the tree structure with the primary key of the table to provide value for the entity instance. Hibernate JPA on the other hand delegates caching with specialized tools [4], whereas in OpenJPA and EclipseLink JPA second level cache is integrated into the core of the framework [6,7]. With an undeniable advantage of JPA framework, however, the problem exists with "Over Caching". Maintaining balanced caching can repair defects, but to improve the quality we often need a third-party framework, which might increase the overall complexity of the architecture [3].

5.2 JPA Frameworks

Our JPA implementations have two major goals: (1) to achieve the best potential performance with the same set of optimization features and capabilities for analysis of the application programs, incorporating program start-up phase and steady phase to reduce the compilation burden and computational cost, and (2) to provide a base implementation for JPA providers to affix practical and effective information collections. To achieve these goals, we selected specific tasks and methods to compile, using operations similar to those discussed in Chapters 3 and 4 that show the practical results of JPA APIs.

- Hibernate Designing Model: Hibernate JPA is a specialized ORM mapping architecture that determines binary Many-One and Many-Many relationships, with the indirect support of entities to map tables in database [4]. Being JPA compliant, Hibernate assists the exclusive JPA API classes such as the EntityManager and EntityManagerFactory objects to perform persistence. Figure 5.1 describes the persistence.XML file with properties such as batch size, fetch size and connection pool size with connection flushes and refreshes, to sync the status with the database. According to [31], flush and refresh altered the persisting objects before the close() method. A second-level cache was configured on a per-class basis using org.hibernate.cache.CacheProvider jar and query results were cached with updated and parameterized queries.

Figure 5.1: Hibernate JPA Persistence.XML file:

- EclipseLink JPA Designing Model: To build a high-performance API in EclipseLink JPA, with automated mapping for metadata, we used pessimistic and optimistic locking, which improved the code adaptability and avoided deadlocks [7]. With the persistence.XML file for database persistence, a major concern was the problem of stale data which was avoided by the @Optimistic [7] locking policy. We replaced the CHANGED COLUMNS [7] policy to ensure that columns were updated and persisted into the database. We used object caching to avoid duplication of objects at the cache level, which improved database quality in data-type conversions [7]. The extensible unit cache (L1) and isolated persistence context cache (L2) were refreshed in coordination with updates providing concurrency with stale data. Figure 5.2 shows an EclipseLink JPA persistence.xml file with properties assigned to the JPA API including batch size, fetch size and connection pool size.

Figure 5.2: EclipseLink JPA Persistence.XML file:

- OpenJPA Designing Model: To manage complex JPA API and compatibility issues in OpenJPA, we used features such as second level caching with a schema tool to create applications with intrinsic sets of classes and interfaces. As a JPA complaint architecture, the graphs and trees with object-oriented operations, like inheritance, were handled with strict standard specification [6]. Another improvement was to monitor the process with auto-generate sub-classes and byte-code weaving tools [6] to dynamically generate a graph model at run-time. Although creating sub-classes is not recommended, we used it for dynamic byte-code enhancements to

perform join and subjoin queries [6]. A reverse mapping concept was used to outline comprehensive meta-data [6] from an existent database schema model, and meet-in-the-middle architecture was automatically applied to capture outstanding exports of the schema in an XML file. A java org.apache.openjpa.jdbc.schema.Schema Tool -a reflect -f schema.XML [6] further can be used to manage the sub-classing enhancements by reducing the cost of the application. Figure 5.3 shows an example of the OpenJPA persistence.xml batch size, fetch size, connection pool size and other properties assigned in the JPA API with RuntimeUnenhanced for classes [6].

Figure 5.3: Open JPA Persistence.XML

- DataNucleus JPA Designing Model: In a DataNucleus JPA, API features such as UUID value generators, and two-level caches for fields loaded from the cache rather than the database to configure meta-data and manage relations using annotation [20] improved the performance. Although the fetch plan for JPQL queries was used with a schema and enhancement model for difficult N-1 unidirectional joins, a fallback connection pool was applied to override the memory management for distributed threads to retain transaction read/write and handle complicated implicit joins [20]. A transparent persistent architecture for meta-data helped auto-manage byte-code enhancement [20]. The schema tool generated a schema with validation, using the persistence.XML file properties are shown in Figure 5.4. We used a datanucleus.schema.autoCreateDatabase property for schema creation,

and an ehcache cache with a datanucleus.cache.level2.type property and jar file performing column based caching [20]. We created a thread safe model to manage datanucleus.Multithreaded properties. Although EntityFactoryManager created a single thread model, we used the datanucleus.Retain property to clear the values generated in the previous transaction with roll-backs [20].


```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.1" xmlns="http://xmlns.jcp.org/xml/ns/persistence" xmlns:xsi="http://
    <persistence-unit name="EclipseLink" transaction-type="RESOURCE_LOCAL">
    <provider>org.datanucleus.api.jpa.PersistenceProviderImpl</provider>
    <exclude-unlisted-classes>false</exclude-unlisted-classes>
<properties>
 <property name="javax.persistence.jdbc.driver"  value= "oracle.jdbc.OracleDriver"/>
 <property name="javax.persistence.jdbc.url"  value= "jdbc:oracle:thin:@localhost:1521:xe"/>
 <property name="javax.persistence.jdbc.user"  value= "test"/>
 <property name="javax.persistence.jdbc.password"  value= "test"/>
 <property name="datanucleus.schema.autoCreateAll" value="true"/>
 <property name="datanucleus.schema.validateTables" value="false"/>
 <property name="datanucleus.schema.validateConstraints" value="false"/>
 <property name="datanucleus.cache.level2.batchSize" value ="11500"/>
 <property name="datanucleus.query.useFetchPlan" value="true"/>
 <property name="datanucleus.query.flushBeforeExecution" value="false"/>
 <property name="datanucleus.rdbms.statementBatchLimit" value="7500"/>
<property name ="datanucleus.connection.nontx.releaseAfterUse" value="false"/>
<property name="datanucleus.enableStatistics" value="true"/>
<property name="datanucleus.cache.collections" value="true"/>
<property name="datanucleus.cache.level1.type" value="soft"/>
<property name="datanucleus.cache.level2.type" value="ehcacheclassbased"/>
<property name="datanucleus.cache.level2.mode" value="ALL"/>

</properties>
</persistence-unit>
```

Figure 5.4: DataNucleus JPA Persistence.XML

5.3 Experiment Design

5.3.1 Evaluation of JPA frameworks

To ensure accuracy, the experiment was divided into two main stages. In the first stage, we conducted an event execution analysis in two phases (start-up phase and steady phase) that reviewed information through an initial 30 execution per phase per query. In the second stage, we compared the results generated in the previous phase and performed 30 more execution with the correlation analysis, using optimal JPA properties defined in persistence.XML file to improve the confidence of the results to 95%. The CRUD operation in second analysis followed the same protocols defined in the first stage. The second study allowed us to conduct statistical analysis with a high confidence level of 95% with JPA benchmarks, as indicated in Tables 5.1, 5.2, 5.3 and 5.4. The correlation coefficient matrix found high and low correlation among components, as shown in Tables

Table 5.1: Hibernate JPA Benchmarks

Query	JPA Provider	Benchmarks	Min Heap
Aggregate Query	Hibernate	fetch size value="150"	100MB
Join Query		batch size value="20"	100MB
		minimum HeapSize = 200	80MB
Sub-Query		maximum Heap Size upto 5 times	75MB
		hibernate.c3p0. maximum	100MB
		hibernate.c3p0. maxsize	125MB
		hibernate.c3p0 .min value="7"	125MB

Table 5.2: EclipseLink JPA Benchmarks

Query	JPA Provider	Benchmarks	Min Heap
Aggregate Query	Eclipselink	fetch size value="150"	100MB
Join Query		batch size value="20"	100MB
		minimum HeapSize = 200	80MB
Sub-Query		maximum Heap Size upto 5 times	75MB
		eclipselink.c3p0. maximum	100MB
		eclipselink.c3p0. maxsize	125MB
		eclipselink.c3p0 .min value="7"	125MB

5.5, 5.6, 5.7 and 5.8. The tasks were divided into three major categories: JOIN, SUB-QUERIES, and AGGREGATIONS.
Analyzing a JPA query-driven API with timer based execution is technically difficult, due to the abstract classes, objects and interfaces used to perform asynchronous callback to metadata. Performance and feature optimization suggested in [31, 32] solved some of the challenges, by highlighting the features and performing comparative analysis on local databases. Due to difficulty managing the raw data set we semi-automated the JPA API analyses on local databases, then performed dynamic data analysis with some constraints, using JPA benchmarks and annotations to achieve a statistical solution. The objective was to examine JPA benchmarks and find correlation among them.

- Start-up Phase: To measure the accuracy of the application we first divided the analysis into phases: start-up and steady. The start-up phase was sub-divided into three main components, to measure the event execution through JVM and GC cycles, the basic creation/insertion into the database and the initial java classes and interfaces, including EntityManagerFactory and EntityManager objects. The steady phase does not include initial java compilation files. To measure the start-up performance we followed two defined steps:
  1.)Measure the execution time of the initial java operations, along with the creation/insertion into the database for every entity in JPA API and loading initial java library. The results in p measurements were x_ij and j=1 with 1 $\leq$i $\leq j$.
  2.)Calculate the results with two scaling factors CPU and GC cycle using standard Deviation (SD) and average mean, with an appropriate sample size of 480,000 entries.

- Steady Rate: The steady phase of the analysis was driven by long-running (select) queries, measured using the total execution time with the start-up phase excluded. Low variability and less variance in the data due to a strict sample size allowed us to perform the steady phase analysis. The main question, however, was when should the steady state start? For a long-running query in JPA API that requires pulling or pushing data while executing the SQL operations, the steady phase starts with the event execution, excluding the basic create and insertion operations. The process was repeated 30 times for each query per JPA benchmarks, and by using the large datasets as inputs we performed all 16 queries in a timer based environment until the execution was complete. To quantify the steady-state performance, we calculated p executions with an at-most q benchmark and a k

Table 5.3: Open JPA Benchmarks

Query	JPA Provider	Benchmarks	Min Heap
Aggregate Query	OpenJPA	fetch size value="150"	100MB
Join Query		batch size value="20"	100MB
		minimum HeapSize = 200	80MB
Sub-Query		maximum Heap Size upto 5 times	75MB
		openjpa.jdbc.DB Dictionary value=batchLimit	100MB
		openjpa. Connection.RetainMode value="always"	125MB
		Openjpa.scale value="7"	125MB

measurement per execution.

1) Execution i determined by s_i is considered to be the steady-state performance [48]. It is based on the coefficient of variance or standard deviation of the k iterations (s_i- k to s_i), with a threshold between 0.05 or 0.10 to assess the confidence of 95% in the results. 2) The executed queries were computed with the mean x_i of k as the benchmark in the steady phase operations. The confidence levels for different executions in all four JPA implementations were the calculated, with 95% confidence over the x_i mean measurements. The initially computed means across multiple JPA implementations were calculated for every execution i, and the calculated confidence to retain autonomy of the measurements for every iteration was (s_i- k to s_i), with a threshold between 0.05 or 0.10.

$$\bar{x_i} = \sum_{j=s_i-k}^{s_i} x_{i_j} \tag{5.1}$$

Table 5.4: DataNucleus JPA Benchmarks

Query	JPA Provider	Benchmarks	Min Heap
Aggregate Query	DataNucleus	fetch size value="150"	100MB
Join Query		batch size value="20"	100MB
		minimum HeapSize = 200	80MB
Sub-Query		maximum Heap Size upto 5 times	75MB
		datanucleus.connection. PoolingType value="DBCP"	100MB
		datanucleus.autoStart Mechanism value="None"	125MB
		datanucleus.cache.level2 .maximum value="100"	125MB
		datanucleus.cache.level2 .maxSize value="500"	125MB

5.3.2 Benchmarks And Machine

Tables 5.1, 5.2, 5.3 and 5.4 show some of the major benchmarks used in the statistical analysis. The benchmarks were based on JPAB [46], which is a JPA based benchmark suite under GNU license, consisting of several benchmark tests with indexing, basic and threading. We used GC and CPU for scaling data with the benchmarks, with an initial heap size of 120 MB. The open source benchmark suite [46] version we used was released in 2012. A medium size experiment with a limited heap size range was conducted using up to six times periodical increments . Local persistence used Oracle 12c and Visual VM to capture GC and CPU cycles in all four JPA implementations: Hibernate 5.2.3 Final, EclipseLink 2.6.4, OpenJPA 2.4.1 and DataNucleus 5.0 with JPA 2.1 on a Pentium 4 processor running Windows 7. The operations performed on the implementations reached 1 GB in memory consumption, and persisted databases in single location with up to 480,000 entries.

5.3.3 Event Execution with Start-up Rate and Numerical Results

To measure the execution time of events among JPA benchmark, a warm-up set of Java class and objects, including 480,000 insert entries, took a minimum of 680 milliseconds, with a maximum of 2,370 milliseconds with default JPA properties in Hibernate JPA. DataNucleus JPA API took a total minimum time of 1,045 milliseconds, with a maximum of 13,724 milliseconds with default JPA settings, followed by OpenJPA which was worse taking approximately 892 milliseconds, and a maximum of 26,475 milliseconds. EclipseLink JPA showed a moderate performance with a total minimum time of 532 milliseconds and a maximum of 1,274 milliseconds to finish the execution (an average response time for select queries).

The box-plot graph in Figure 5.5 and Figure 5.6, show the mean values, including whiskers and outliers, of every implementation performing real-time start-up operations. The values in Quartile 1 (Q1) and Quartile 3(Q3), including IRQ= (Q3-Q1), to determine the inter-quartile range, were prepared using the graph median/mean. The solid horizontal line represents the first and third quartile with 50% of the data objects, the thin vertical line above the box indicates outliers with an extreme point up to 1.5 times the inter-quartile range from the mean, and the top and bottom values display the results. The box-plot graph in Figure 5.5 and Figure 5. 6 shows the default fetch size per batch size, based on the CPU and GC cycles in the start-up phase. The normalized scale for the

CPU cycle uses y-coordinates and JPA API implementations on X. The graphs provide the CPU and GC consumption in bytes for fetch size per batch size in the start-up stage. The results of the phase show that Hibernate JPA performed computationally faster than the other implementations. DataNucleus JPA had the worst CPU consumption of up to 38% and heavy outlier detection of 130%, as well as high computational costs, OpenJPA reached as high as 85% with outliers along with EclipseLink JPA at approximately 82%. The CPU cycles in Hibernate JPA was 40% less than DataNucleus JPA, and approximately 10% less than the other two implementations. Similarly, the second graph (Figure 5.5) measured the GC cycles consumed in the startup phase. Figure 5.6 represents a normalized form of GC in bytes and shows that Hibernate JPA performed well with GC cycle with a mean value of 3.07 which was less than EclipseLink JPA, OpenJPA, and DataNucleus JPA. They had a mean value of approximately 4.02, 5.87 and 12.44. The average mean and standard deviation of 30 execution in all four JPA API using cache size, Heaps size, and batch size are shown in more detail in Appendix B (Table B.1, B.2, B.3 and B.4).

5.3.4 Event Execution with steady-up Rate and Numerical Results

The steady-state analysis is based on long-running queries, which are measured using the total running time, excluding start-up. To improve the measurement process, we estimated query execution using JPA benchmarks over the execution period, or until the query executes and displays the result on the console. With the low variance of the p execution values, at-most q benchmark and a k measurement were calculated per execution.

1) The execution i in steady state s_i, is based on the standard deviation of k measurements per execution, using the same threshold as start-up.

2) To executed queries, the mean x_i for k benchmarks per invocation or execution was calculated to determine a correlation, using the JCTs measuring technique and Visual VM tools.

The initial computation of steady phase used 30 execution per query using JPA benchmarks per execution i, and calculated coefficient of variance per JPA benchmark. With low run-time variability, the coefficient of variance or standard deviation had a significant effect on the evaluation. Measuring standard deviation while maintaining the threshold at approximately 5% per JPA benchmark, with maximum/minimum

generalities of 10% in start-up and 20% in steady phase, provided approximate results. Finally, the boxplot graphs in Figures 5.7 and Figure 5.8 depict steady performance, with small variations in the Q1/Q3 mean. The statistical test was based on the hypothesis that a steady phase provides a better confidence in the results for JPA APIs, for conducting a meaningful JPA statistical analysis with consideration towards metadata mapping. To understand the proportionality among JPA benchmarks, Big 'O' asymptotic notation and the correlation analysis increased the confidence in the results in the second phase. For example, DataNucleus JPA with a CPU and GC cycles in Figure 5.7 and Figure 5.8 showed heavily skewed mean compared to other JPA API implementations and consumed memory in the range of 200 MB to 1200 MB

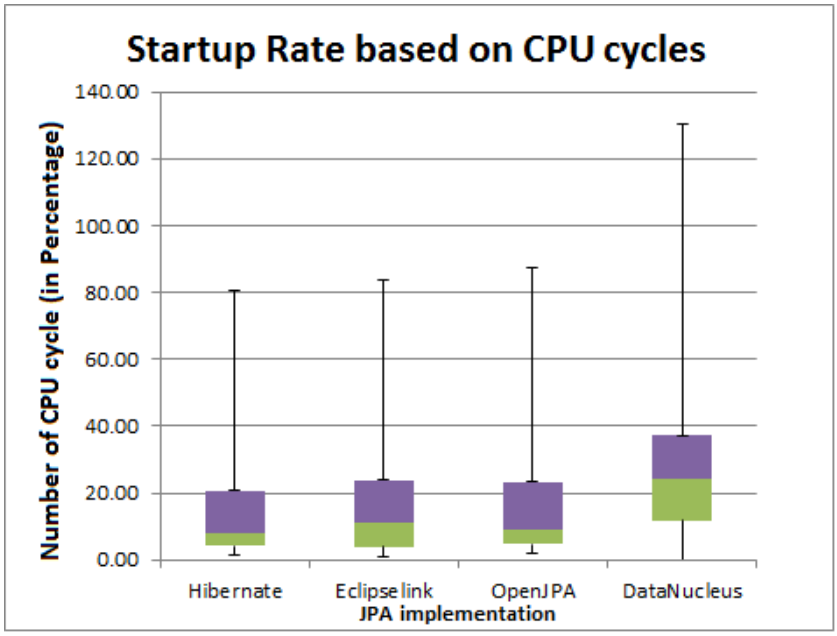

Figure 5.5: First Stage (Start phase) results using CPU cycle as scaling factors for JPA benchmarks (fetch size per increasing batch size) on y-axis and JPA implementation on X-axis

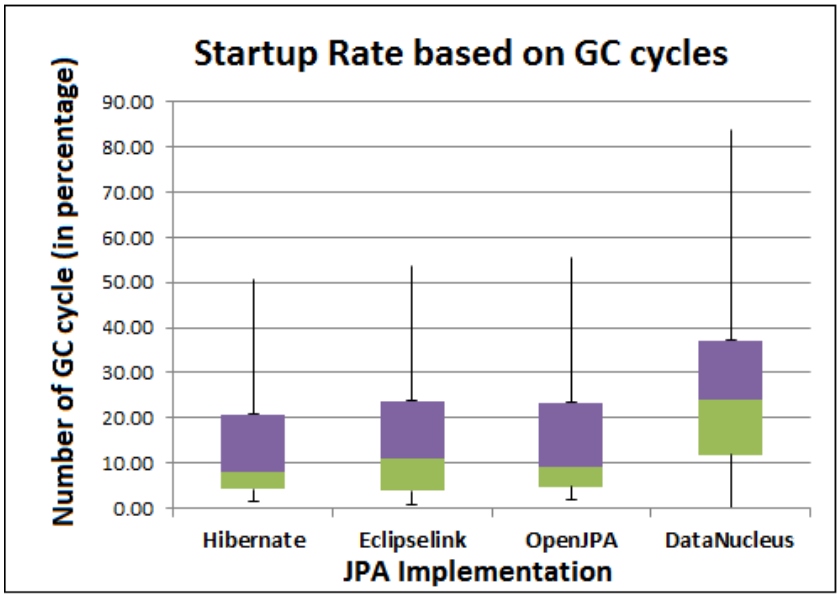

Figure 5.6: First Stage (Start phase) results using GC cycle as scaling factors for JPA benchmarks (fetch size per increasing batch size) on y-axis and JPA implementation on X-axis

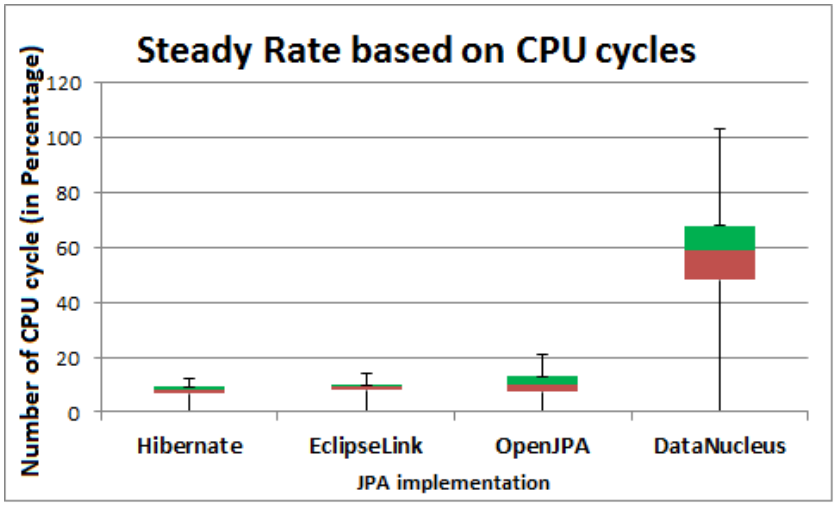

Figure 5.7: First Stage (Steady phase) results using CPU cycle as scaling factors for JPA benchmarks (fetch size per increasing batch size) on y-axis and JPA implementation on X-axis

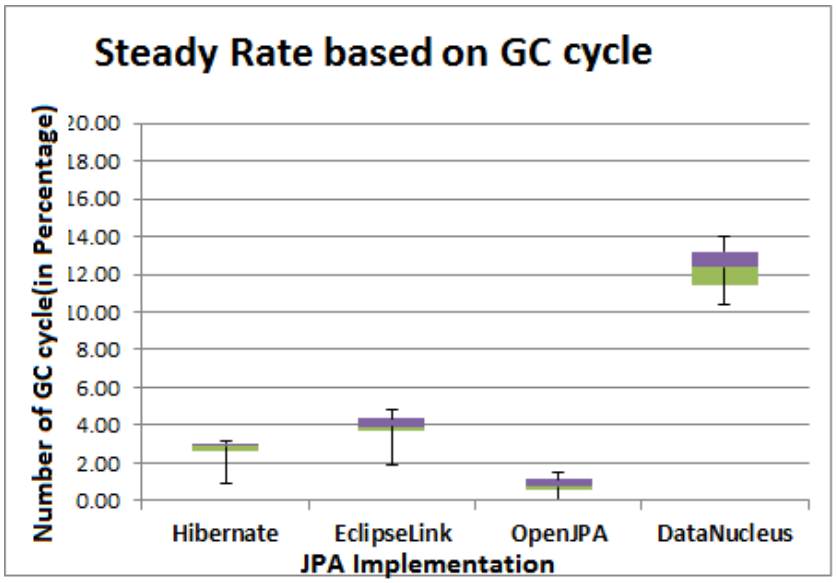

Figure 5.8: First Stage (Steady phase) results using GC cycle as scaling factors for JPA benchmarks (fetch size per increasing batch size) on y-axis and JPA implementation on X-axis

Table 5.5: Pearson's Correlation Coefficients-Hibernate JPA Implementation

Heap Used	-					
Heap Size	0.38	-				
Class Loaded	0.36	0.89	-			
Batch Size	0.72	0.67	0.27	-		
Fetch Size	0.86	0.53	0.53	-0.21	-	
Cache Size	0.16	0.90	0.44	-0.11	0.23	-
Metrics	Heap Used	Heap Size	Class Loaded	Batch Size	Fetch Size	Cache Size

Table 5.6: Pearson's Correlation Coefficients-EclipseLink JPA Implementation

Heap Used	-					
Heap Size	0.38	-				
Class Loaded	0.77	0.54	-			
Batch Size	0.44	0.34	-0.16	-		
Fetch Size	-0.26	0.21	0.11	0.99	-	
Cache Size	0.11	-0.9	0.13	-0.17	0.21	-
Metrics	Heap Used	Heap Size	Class Loaded	Batch Size	Fetch Size	Cache Size

Table 5.7: Pearson's Correlation Coefficients-Open JPA Implementation

Heap Used	-					
Heap Size	0.46	-				
Class Loaded	0.19	0.37	-			
Batch Size	0.84	0.79	0.26	-		
Fetch Size	0.38	-0.62	0.29	0.96	-	
Cache Size	-0.16	0.37	0.25	0.16	-0.29	-
Metrics	Heap Used	Heap Size	Class Loaded	Batch Size	Fetch Size	Cache Size

Table 5.8: Pearson's Correlation Coefficients-DataNucleus JPA Implementation

Heap Used	-					
Heap Size	0.98	-				
Class Loaded	0.84	0.77	-			
Batch Size	0.87	0.97	0.53	-		
Fetch Size	0.26	0.22	-0.16	0.51	-	
Cache Size	0.56	0.96	0.15	0.76	0.34	-
Metrics	Heap Used	Heap Size	Class Loaded	Batch Size	Fetch Size	Cache Size

to execute simple Join operation. This heavy increase in GC cycle suggested memory leaks that lead to OutofMemory, but our assumption was incorrect because the newly added memory consumption was due to lazy loading and incorrect meta-data creation, which significantly increased meta-data size. Persistence in DataNucleus JPA uses an enhancer tool and schema tool that resulted in high complexity and such exceptions. Hibernate JPA compared to other implementation performed well with CPU and GC

cycle both. An exceptional result was observed with OpenJPA using compile time weaving and a similar result was depicted with EclipseLink JPA with GC cycle.

5.4 Final Analysis

To increase the confidence in the results of the JPA API using JPA benchmarks, we performed a second analysis with 30 executions of the same queries. In the previous stage, we measured the mean and coefficients of variance using CPU and GC as scaling factors on the y-axis, and JPA benchmarks were averaged into JPA implementation on the x-axis. We used boxplot graphs in the first stage to show the analysis per implementation over start-up and steady phase, we used the similar approach to derive correlation among JPA benchmarks in this stage and showed a high correlation among JPA benchmark with low variance.

A 95% confidence was achieved, followed by direct and indirect proportional relationships among the benchmarks. Figure 5.9, Figure 5.10, Figure 5.11, and Figure 5.12 show the mean over 30 executions per JPA benchmark over GC and CPU, with overlapping results. A similar observation was measured over CPU cycles with high variance and low confidence in DataNucleus JPA, as shown in Figure 5.9 and Figure 5.10 using increasing Heap Used and Batch size as JPA benchmark. However, Hibernate JPA had a lower variance of 0.24%, compared to EclipseLink JPA with 0.52% and OpenJPA with 0.37%. A major reason for 30 executions per JPA benchmark was to investigate fluctuations in the steady period and to understand how fresh executions are affected by second-level cache using counter instance. JPA benchmarks included memory consumption and batch size while other JPA benchmarks are shown as tables in Appendix B. Over the entire execution process, Hibernate JPA completed one execution cycle in a minimum of 279.9 seconds and a maximum of 306.77 seconds, using at least 600 MB per heap memory on average, and an optimal batch size of 10,500. A similar observation was conducted for EclipseLink JPA, OpenJPA and DataNucleus JPA.

We observed heavy computational cost and high memory consumption in DataNucleus JPA, using 4096 MB Heap size. The execution process in DataNucleus JPA took an average 279.9 seconds, with a maximum rate of 306.77 seconds and average heap size of 2,600 MB with +-30% variation. However, EclipseLink JPA ranged +-10% from the mean value with low memory consumption compared to other implementations. In OpenJPA, the execution cycle ranged between 597.9 seconds and 405.2 seconds, using a minimum heap size of 1255 MB. OpenJPA performed worse than DataNucleus JPA,

with the highest fluctuation of +-44% from the mean. In next couple of subsection, we explained the startup and steady phase performance and a correlation matrix among JPA benchmarks.

5.4.1 Final Start-up Results

To retrieve the start-up results in the second analysis, we performed a full-scale data analysis of GC and CPU cycles, with the initial start-up phase including loading, and basic insertion and creation operations. On average, the results showed a confidence of 95%, with the suggested fluctuation of 1.8% to 4.3% over/below the threshold in Hibernate JPA. Other implementations, such as DataNucleus JPA, showed high computational cost and fluctuation in the results, and varying thresholds for heap used per GC and CPU cycle of approximately 7.3% to 12.6%.
The graph based on CPU cycle in the previous stage showed the results in the startup phase using initial five to ten executions, followed by a final review of 30 executions per JPA API (an average of the JPA benchmarks). To improve the confidence and accuracy per execution of the results generated by the JPA implementations, those that favored Hibernate JPA, DataNucleus JPA or OpenJPA were compared. The results demonstrated through 30 execution per query showing that Hibernate JPA is better than DataNucleus JPA in memory consumption, though this assumption was suspected. OpenJPA and EclipseLink JPA showed overlapping results which was difficult to analyze. So, to develop more reasonable results, we computed both overlapping/non-overlapping confidence intervals [48]. A possible solution to overlapping results was by improving the sample size and mean; this was simple and reduced the number of misleading results. The results showed many interesting observations:

- The methodology we used proved to be exceptional at handling fractional data. The total range of comparisons and data generated was overlapping, which made the computation process complicated and difficult to assess that one JPA implementation was better than another.

- For a reasonable number of measurements, the parameters in the statistical methodology led to correct conclusions without overlapping confidence with comparisons of the threshold between 1% to 3%, compared to over 12.6% in DataNucleus JPA.

- We observed that the start-up phase JPA benchmarks in some implementations performed better (e.g.batch sizes of 1,000 in Hibernate JPA). The accuracy of the

mean and standard deviation was also improved by using consistent sample size with parameters in persistence.XML files as stated in section 5.2.

- An intriguing observation was the autonomy of observations and optimization of results using feature analysis in [31].

5.4.2 Final Steady Results

A JPA API typically consists of hundreds of abstraction layers (e.g. classes, objects), and using the data over a steady period to conduct an analysis is difficult. A common approach to overcoming this complexity for statistical analysis is to summarize the components. Unfortunately, the simple techniques to measure skew and asymmetry are usually unbounded and not easy to apply to an application, due to non-deterministic factors and unprecedented benchmarks. To manage this, we deployed higher-order statistics to catch key features of the data that adequately summarize the metrics, to simplify the correlation among varying components. To determine confidence in the results of 30 executions, we performed JPA implementations using every JPA benchmark required in a JPA API. Figure 5.9, Figure 5.10, Figure 5.11 and Figure 5.12 show the mean of the 30 iterations per JPA benchmark (Heap Used and Batch size) with 95% confidence over CPU and GC cycle percentages, using batch sizes from 500 to 60,500 and Heap size of minimum 120 MB and up to 270 MB. Since batch size and memory consumption directly affects performance and execution of the JPA API, they were used on the x-axis to determine the results. A correlation matrix was created among JPA benchmarks as shown in Table 5.5, 5.6, 5.7 and 5.8. The JPA benchmarks with high correlation have a directly proportional relationship among them whereas the negative correlation was observed as an inversely proportional relation. For components with positive correlation, we create a graph as shown in Figure 5.13 for Hibernate JPA, Figure 5.14 for EclipseLink JPA, Figure 5.15 for OpenJPA and Figure 5.16 for DataNucleus JPA).

- We measured JPA API correlation at 95% confidence, followed by proportionality relationships between different JPA benchmarks. Hibernate JPA benchmarks, such as cache size and fetch size, showed strong correlation with heap used and heap-size (0.86% and 0.90% respectively) compared to DataNucleus JPA (0.45%), OpenJPA (0.37%), and EcliplseLink JPA which had a negative correlation (-0.9%) (Figure 5.13, Figure 5.14, Figure 5.15 and Figure 5.16). Metric extraction in JPA API

implementation took O(n), using a linear scale. The strong positive correlation of Hibernate JPA showed that its cache size is directly proportional to the heap used, and the processing time (in ms) is also proportional to the cache size, which means an increase in cache size would definitely affect the heap usage as shown in Figure 5.13.

- In every JPA implementation, cache size and fetch size had weak correlation (0.23 in Hibernate JPA, 0.21 in EclipseLink JPA, -0.29 in OpenJPA and 0.34 in DataNucleus JPA). Tables 5.5, 5.6, 5.7 and 5.8 show that cache size is inversely related to fetch size. This observation stated that an increase in the batch size and fetch size would not improve the performance of the JPA API.

- Observations of batch size and memory consumption (i.e. heap used) in Table 5.5, 5.6, 5.7 and 5.8 showed strong positive correlation with Hibernate JPA (0.72%), DataNucleus JPA (0.87%) and OpenJPA (0.84%), with only EclipseLink JPA (0.44%) showing indirect correlation between batch size and memory consumed in the implementation. The strong positive correlation of JPA API proves that batch size is directly proportional to heap used, which means an increase in batch size affects heap usage. For such situations, we used high batch sizes.

5.5 Conclusion

In this chapter, we proposed a manageable technique for reiterating statistical analysis in existing JPA implementations using local databases. The intention was to build and capture the start-up and steady rate, based on CRUD queries and transient persistence processes.

- Different JPA benchmarks with 16 major SQL queries and other CRUD operations were conducted using JCT framework and Visual VM parameters troubleshooting the APIs results using start-up and steady phase.

- For each JPA API, we calculated the strong and weak correlations among JPA benchmarks (Correlation Matrix).

- A confidence of 95% was obtained in the analysis while retrieving results from the JPA APIs using two stage analysis.

- Hibernate JPA showed better result compared to other JPA implementations in memory consumption per number of classes called. In EclipseLink JPA results showed high correlation between memory consumption and fetch size per number of classes and batch size.
- Similarly, memory consumption and batch size showed positive correlation in OpenJPA, while in DataNucleus JPA memory size was directly proportional to the memory consumed.

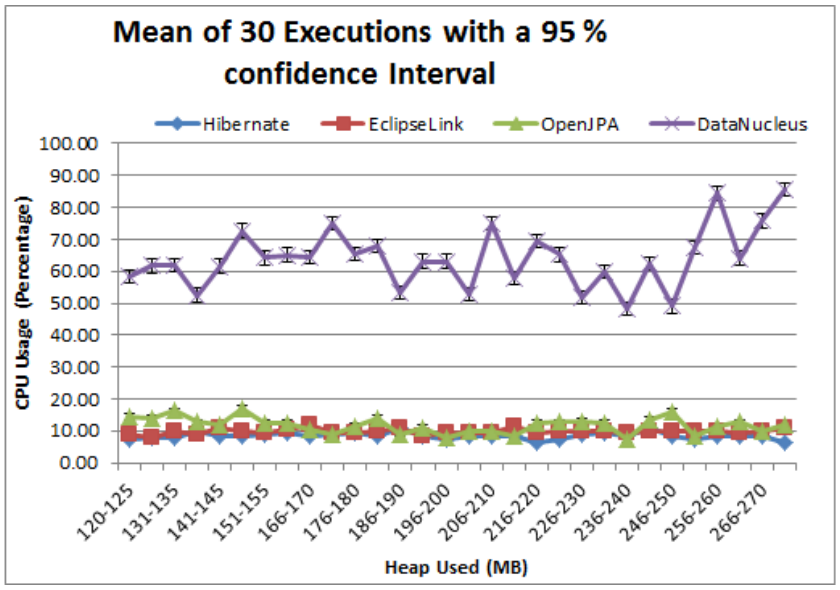

Figure 5.9: Second stage (Start phase) Mean of 30 iteration with 95% confidence in results per JPA benchmarks with CPU cycle on y-axis and increasing JPA Benchmarks on X-axis

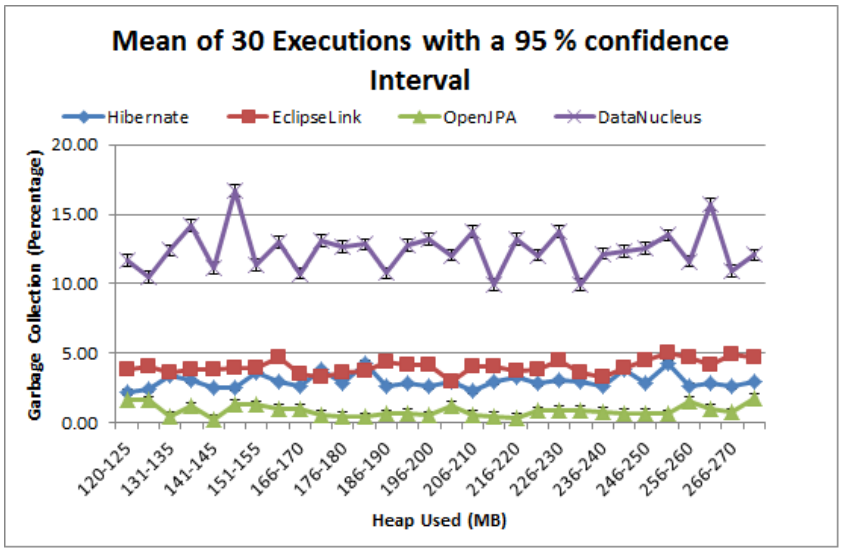

Figure 5.10: Second stage (Start phase) Mean of 30 iteration with 95% confidence in results per JPA benchmarks with GC cycle on y-axis and increasing JPA Benchmarks on X-axis

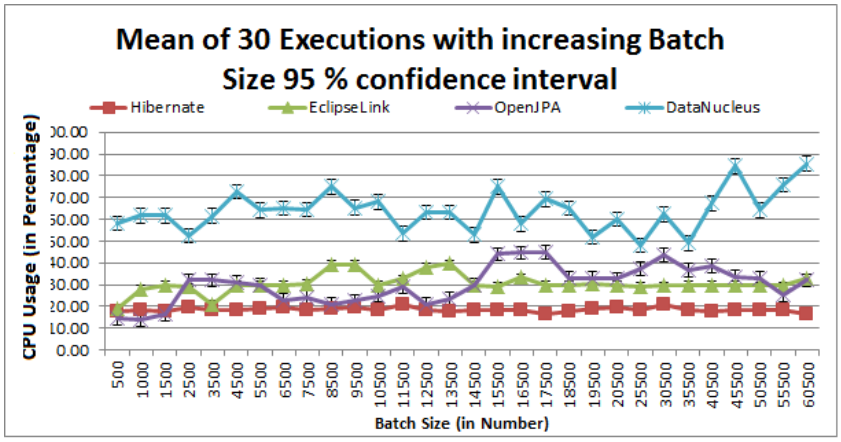

Figure 5.11: Second stage (Steady phase) Mean of 30 iteration with 95% confidence in results per JPA benchmarks with CPU cycle on y-axis and increasing JPA Benchmarks on X-axis

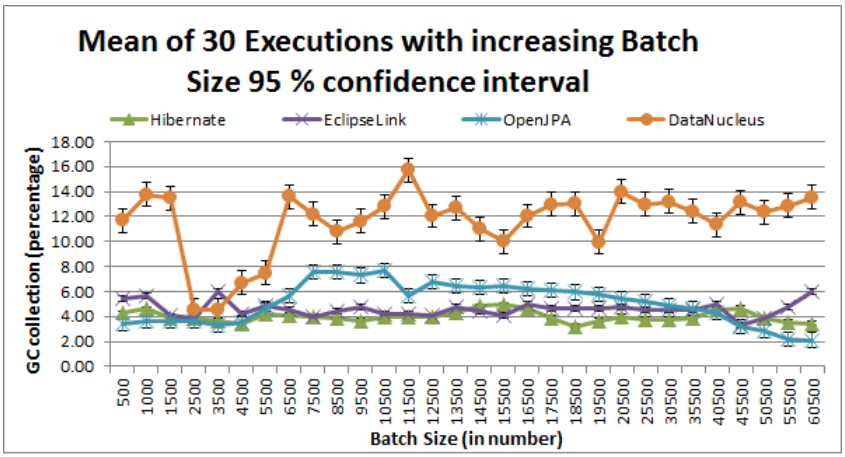

Figure 5.12: Second stage (Steady phase) Mean of 30 iteration with 95% confidence in results per JPA benchmarks with GC cycle on y-axis and increasing JPA Benchmarks on X-axis

Finally, Figure 5.13 includes three graphs using Hibernate JPA that show batch size, cache size and number of classes called per memory consumed or Heap Used with high positive correlation. Similar results in Figure 5.14 for EclipseLink JPA showed memory consumed per number of classes and fetch size per batch size over a steady period with positive correlation. OpenJPA in Figure 5.15 showed fetch size per batch size and memory consumed per batch size, while DataNucleus JPA in Figure 5.16 showed memory size per memory consumed and memory consumed per batch size. This suggests that cache size and fetch size affect the performance of JPA API, they are correlated, and they contribute toward improving JPA API.

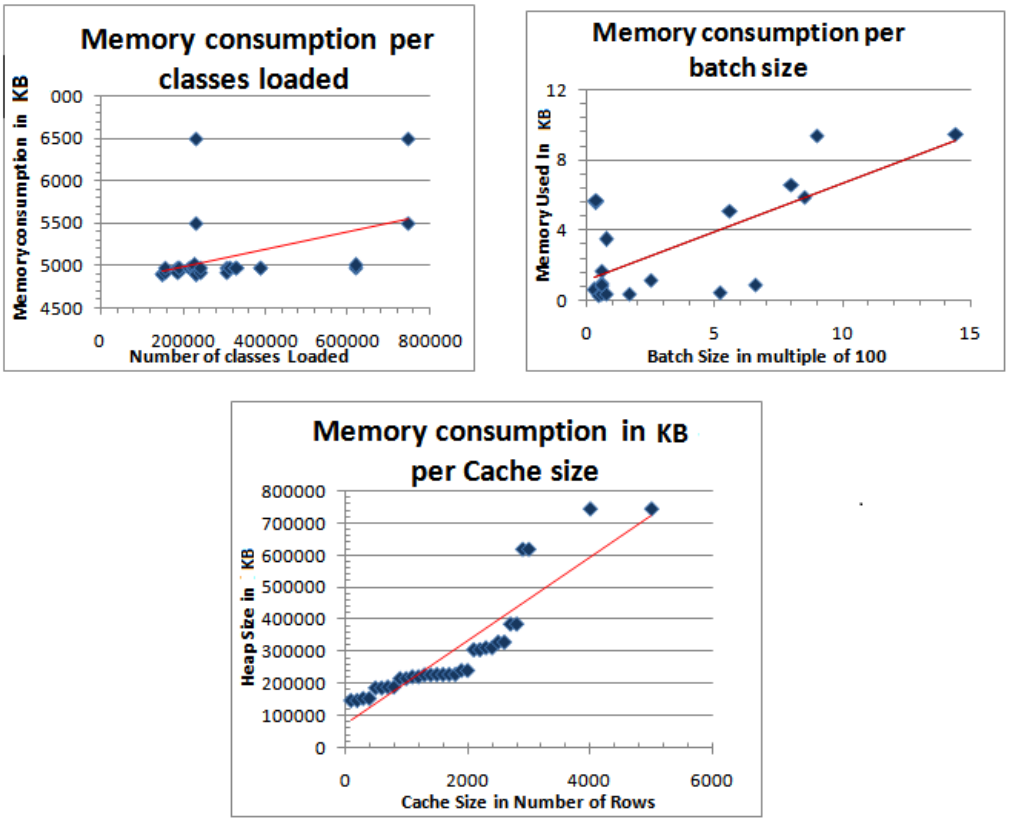

Figure 5.13: Pearson's Correlation coefficient results for Hibernate JPA with positive correlation between JPA Benchmarks

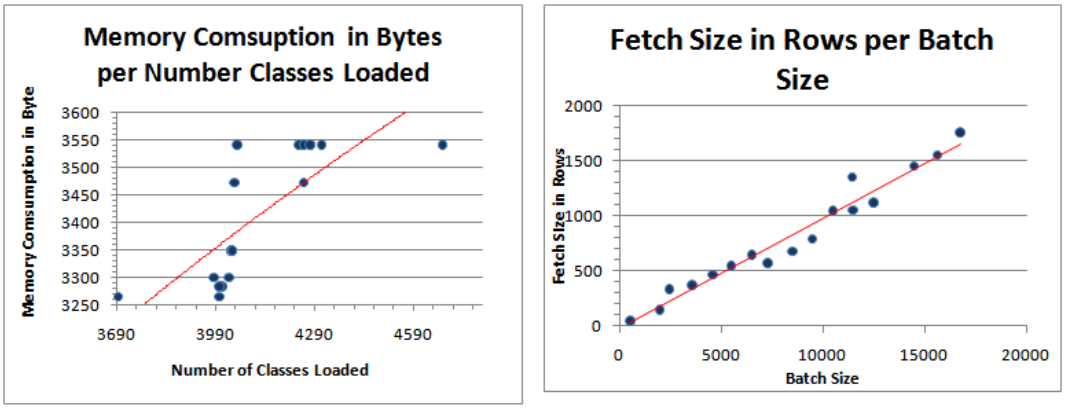

Figure 5.14: Pearson's Correlation coefficient results for EclipseLink JPA with positive correlation between JPA Benchmarks

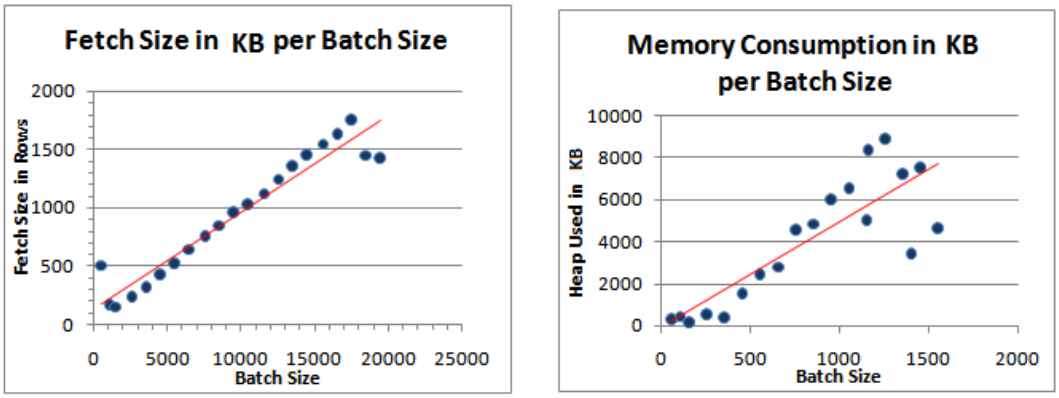

Figure 5.15: Pearson's Correlation coefficient results for OpenJPA with positive correlation between JPA Benchmarks

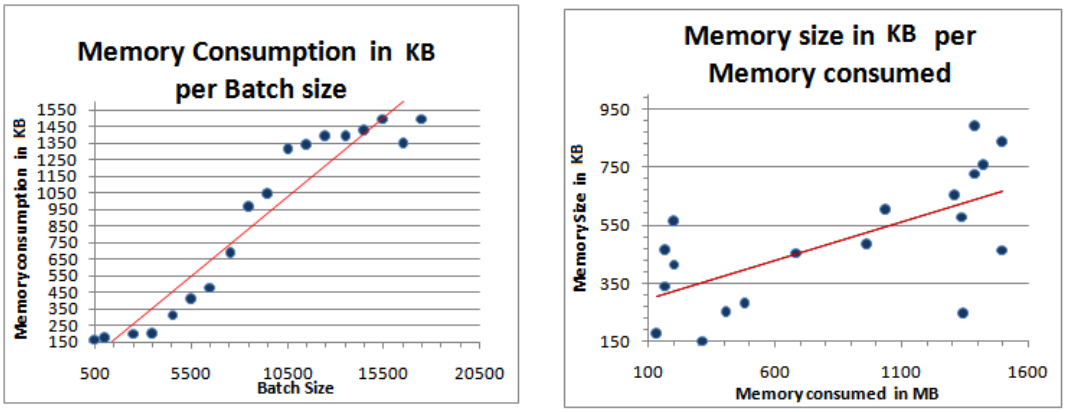

Figure 5.16: Pearson's Correlation coefficient results for DataNucleus JPA with positive correlation between JPA Benchmarks

Chapter 6

Conclusion and Future Work

6.1 Conclusion

In this chapter, we present a review of our research contribution, as well as concluding remarks and proposed directions for future work. After several proposals based on JPA and JPA implementation, persisting databases is still an active topic in the research community. In an optimal performance, persisting data will add incremental changes to existing JPA implementations. As persisting data on cloud remains highly dependent on the Internet, it needs to interact seamlessly in an uninterrupted area of operation. The sensitivity of cloud databases for existing JPA implementations was clear when we measured the performance compared to local databases. The experiment showed a massive increase in the response time, and CPU and GC instability on cloud.

In this thesis, we initially studied existing JPA high-level implementation architectures and feature analysis, followed by performance analysis based on using local databases with five major complex SQL queries. Though the performance of localhosts was not sufficient to analysis the implementations, we enhanced the experiment by using databases-as-a-service on a cloud. The basic reason for cloud database analysis was to compare it with local database using complex SQL queries, and to analyze the effect of JPA API on the number of CPU and GC cycles required to perform an operation, the number of threads (Live/ Daemon) and the number of classes loaded to execute an event. We considered many features that could enhance the performance of the applications, including parameters such as heap size, batch size and fetch size of the JPA API. The results showed that, overall, local databases outperform cloud databases in all four existing JPA implementations.

We performed statistical analyses of JPA API using local databases, based on startup and steady phase and query execution. The study provided correlation between different JPA benchmarks, and achieved a confidence level of 95% in the results. Although the steady phase was used to differentiate between the initial loading and a real-time execution, the results showed promise. OpenJPA and EclipseLink JPA were affected most by heavy loads when performing operations of 480,000 entries, and the average performance was low. With DataNucleus JPA we observed a significant, and unexpected, degradation of performance and outofmemory exceptions, particularly with heavy workloads.
Considering the results reported in Chapter 5, we concluded that local databases achieved optimal performance without explicitly setting up the heap used, batch size and fetch size. By combining the benchmarks with the execution, we found that the percentage of CPU and GC would be significantly affected if we fixed JPA benchmarks, which have negative correlation. For example, the memory used and memory size in all four JPAs resulted in negative correlation, and adversely affected the performance of JPA API. It can be concluded that Hibernate JPA and DataNucleus JPA are the most advanced of the four, but the performance issues and memory leaks with DataNucleusJPA need to be addressed. OpenJPA has serious computability issues, and EclipseLink hangs and goes into a non-responsive state for most of the operation. The non-responsiveness is not due to memory or compatibility, but internal libraries or structures we weren't able to assess.

6.2 Future work

The following proposed work, which would extend in multiple directions or dimensions, could be considered in the future:

- In the context of existing JPA implementation, it would be constructive to study more implementations with same set of constraints and queries, with different workloads. We could incorporate commercial JPA implementations into the model, to generate results based on complex SQL Queries. We assumed that few critical operations propagated through with limited databases, and including commercial JPA implementations with the queries will force the result set in different directions, and improve the analysis.

- For our JPA persistence analysis we chose 480,000 entries for 16 SQL queries anda limited number of rows for a given experiment, and used MB of data in a real database to study different database parameters. This is an interesting opportunity,

as it could yield non-obvious results for all four implementations, particularly with the commercial cloud or local databases.

- Currently in JPA, both open and commercial providers are performing complex operations to control heavily loaded queries at runtime. Since new and advanced data sources, such as JDO, are involved in implementations, it would be beneficial to reconsider the approach and verify the results of JPA API versus JDO API.

References

[1] K. Ward. CompTIA: More than 90 Percent of Businesses Using Cloud, 2014, Last Accessed Date: 20 June, 2017.

[2] P. C. Linskey, M. Prudhommeaux. An in-depth look at the architecture of an object/relational mapper. In Proceedings of the 2007 ACM SIGMOD international conference on Management of data, 2007, Pages: 889-894, Last Accessed Date: 20 June, 2017.

[3] J. E. Lascano. JPA implementations versus pure JDBC, Congreso de Ciencia y Tecnologa ESPE-2008, Quito-Ecuador, 2008, Pages : 1-10, Last Accessed Date: 20 June, 2017.

[4] Doctrine of Objects, Available At: http://docs.doctrine-project.org/projects/doctrine-orm/en/latest/reference/working-with-objects.html, 2015, Last Accessed Date: 20 June, 2017.

[5] Hibernate. Hibernate JPA Advanced Features Documentations, Available At : http://www.hibernate.org, 2017, Last Accessed: 20 June, 2017.

[6] OpenJPA. OpenJPA Manage Features Documentation , Available At : http://openjpa.apache.org/, 2017, Last Accessed Date: 20 June, 2017.

[7] EclipseLink Concepts Guide. EclipseLink Understanding EclipseLink 2.5, /new-blockhttp://www.eclipse.org/eclipselink/documentation/2.5/, 2013, Last Accessed Date: 20 June, 2017.

[8] Apache OpenJPA. Apache Open JPA 2.0 User's Guide , Available At: http://openjpa.apache.org/builds/1.0.0/apache-openjpa-1.0.0/docs/manual/index.html, 2013, Last Accessed Date: 20 June, 2017.

[9] O. Probst. Investigating a Constraint-Based Approach to Data Quality in Information Systems, Eidgenossische Technische Hochschule Zurich, Doctoral dissertation, 2013, Last Accessed Date: 20 June, 2017.

[10] M. Enoki, Y. Ozawa, H. Horii, T. Onodera. Memory-Efficient index for cache invalidation mechanism with OpenJPA, In International Conference on Web Information Systems Engineering, 2012, Pages: 696-703, Last Accessed Date: 20 June, 2017.

[11] C. Ireland, D. Bowers. Exposing the myth: object-relational impedance mismatch is a wicked problem, In DBKDA 2015, The Seventh International Conference on Advances in Databases, Knowledge, and Data Applications, 2015, Pages: 21-26, Last Accessed Date: 20 June, 2017.

[12] E. E. Ogheneovo, P. O. Asagba, N. O. Ogini. An Object Relational Mapping Technique for Java Framework, International Journal of Engineering Science Invention, 2013, Pages: 01-9, Vol: 6 , Last Accessed Date: 20 June, 2017.

[13] N. Sharma, P. N. Barwal. Electronic Project Proposal Management System for Research Projects Based on Integrated Framework of Spring and Hibernate, International Journal of Science and Research (IJSR)s, Vol: 4, Issue-5, 2013, Last Accessed Date: 20 June, 2017.

[14] M. Keith, M. Schincariol. Introduction to InPro JPA 2, Apress, 2013, Pages: 1-110, Last Accessed Date: 20 June, 2017.

[15] B. S. Sapre, R. V. Thakare, S. V. Kakade. Design and Application of the Hibernate Persistence Layer Data Report System using JasperReports, International Journal of Engineering and Innovative Technology (IJEIT) , Vol: 1, Issue-5, 2012, Last Accessed Date: 20 June, 2017.

[16] C. Bauer, G. King. Java Persistance with Hibernate, 2006, Pages: 123-302, Last Accessed Date: 20 June, 2017.

[17] B. Vasavi. Hibernate Technology for an efficient business application extension, Journal of Global Research in Computer Science, Vol: 6, Issue-25, 2011, Last Accessed Date: 20 June, 2017.

[18] L. Semebera. Comparison of JPA providers and issues with migration, Masarykova University, Diploma Thesis, 2012, Last Accessed Date: 20 June, 2017.

[19] D. Clarke. EclipseLink, Available At: https://dzone.com/articles/introducing-eclipselink, 2010, Last Accessed Date: 20 June, 2017.

[20] DataNucleus JPA. DataNucleus Features in JPA Documentation, Available At : http://www.datanucleus.com/, 2017, Last Accessed Date: 20 June, 2017.

[21] T. Giunipero. Developing a Java persistence API with the NetBeans IDE and EcliseLink, Available At: http://www.oracle.com/technetwork/systems/ts-5400-159039.pdf, 2016, Last Accessed Date: 20 June, 2017.

[22] L. Shklar and R. Rosen. Web Application Architecture: Principles, Protocols and Practices, John Wiley and Sons, ISDN 0-471-48656-6, 2003, Last Accessed Date: 20 June, 2017.

[23] S. Folino. JPA Set vs List, At Available http://simone-folino.blogspot.ca/jpa-set-and-list-using-jointable.html , 2012, Last Accessed Date: 20 June, 2017.

[24] OpenJPA. What is Enhancement Anyway? In OpenJPA , Available At: http://openjpa.apache.org/entity-enhancement.html, 2017, Last Accessed Date: 20 June, 2017.

[25] J. Tee and A. Jefferson. Slingshot Yourself into DataNucleus 2.1 and JPA 2.0 , Available At: http://www.theserverside.com/tutorial, 2014, Last Accessed Date: 20 June, 2017.

[26] Oracle Contributor. EclipseLink Solutions Guide for EclipseLink Release 2.5, Available At: http://www.eclipse.org/eclipselink/documentation/2.5/solutions/toc.html, 2013, Last Accessed Date: 20 June, 2017.

[27] Hibernate. Multithreading in Hibernate, Available At: http://blog.xebia.com/hibernate-and-multi-threading/, 2017, Last Accessed Date: 20 June, 2017.

[28] K. Serneels. Bulk Fetching with Hibernate, Available At: https://dzone.com/articles/bulk-fetching-hibernate", 2014, Last Accessed Date: 20 June, 2017.

[29] Eclipselink. Eclipselink Flushing, Available At: http://www.eclipse.org, 2017, Last Accessed Date: 20 June, 2017.

[30] Eclipselink Transaction management. Introduction to EclipseLink Transactions. Available At: https://wiki.eclipse.org/, 2014, Last Accessed Date: 20 June, 2017.

[31] N. Dhingra, E. Abdelmoghith and H. T. Mouftah. Performance comparision of ORM based JPA implementations, Proceedings of 2nd International Conference on Computer Science Networks and Information Technology, in Montreal, Canada, ISBN No.: 978819313736915, 2016, Last Accessed Date: 20 June, 2017.

[32] N. Dhingra, E. Abdelmoghith and H. T. Mouftah. Review on JPA Based ORM Data Persistence Framework , ICECS 2016 Toronto, Canada. IJCTE, 9th International Conference on Environmental and Computer Science, 2016, Pages: 451-461, Last Accessed Date: 20 June, 2017.

[33] H. Yousaf. Performance Evaluation of Java Object Relational Mapping, University of Georgia Athens, Master Thesis, 2012, Last Accessed Date: 20 June, 2017.

[34] Oracle. Oracle Report White Paper of Oracle 12 Database, 2013, Last Accessed Date: 20 June, 2017.

[35] R. Bhoyar, N. Chopde. Cloud Computing: Service models, Types, Database and Issues, International Journal of Advanced Research in Computer Science and Software Engineering, Vol: 3, 2013, Last Accessed Date: 20 June, 2017.

[36] E. Gorelik. Cloud computing models , Massachusetts Institute of Technology, Doctoral dissertation, 2013, Last Accessed Date: 20 June, 2017.

[37] D. D. Brno. Database management as a cloud-based service for small and medium organizations, Masaryk University Brno, Faculty of Informatics, Doctoral dissertation, 2013, Last Accessed Date: 20 June, 2017.

[38] K. Donkena and S. Gannamani. Performance Evaluation of Cloud Database and Traditional Database in terms of Response Time while Retrieving the Data , Blekinge Institute of Technology, Master Thesis, 2012, Last Accessed Date: 20 June, 2017.

[39] C. Curino. Relational cloud: A database-as-a-service for the cloud, Massachusetts Institute of Technology. Master Thesis, 2013, Last Accessed Date: 20 June, 2017.

[40] Amazon Relational Database Service (Amazon RDS), https://aws.amazon.com/rds/, 2017, Last Accessed Date: 20 June, 2017.

[41] S. Devijver. The problem with JPA/Hibernate (or the future of ORM), Available At: https://dzone.com/articles/problem-jpahibernate-or-future-orm, 2008, Last Accessed Date: 20 June, 2017.

[42] R. Cortez. How to use JPA correctly to avoid complaints of a slow application, Available At: https://zeroturnaround.com/rebellabs/how-to-use-jpa-correctly-to-avoid-complaints-of-a-slow-application/, 2015, Last Accessed Date: 20 June, 2017.

[43] M. Debnath. Manage Concurrent Access to JPA Entity with Locking, Available At: http://www.developer.com/java/manage-concurrent-access-to-jpa-entity-with-locking.html, 2013, Last Accessed Date: 20 June, 2017.

[44] P. H. Charbonneau and I. Tsagklis. Java Code Geeks, Java 2 Resource Center, JVM Troubleshooting Guide, 2015, Last Accessed Date: 20 June, 2017.

[45] M. Lumpe, S. Mahmud, and O. G. Lumpe. jCT: A Java Code Tomograph. In Automated Software Engineering (ASE), 26th IEEE/ACM International Conference on, 2011 , Pages: 616-619, Last Accessed Date: 20 June, 2017.

[46] Benchmark. JPA Performance Benchmarks (JPAB) DataNuleus, EcliseLink, OpenJPA and Hibernate Benchmarks , Available At : http://www.jpab.org/, 2017, Last Accessed Date: 20 June, 2017.

[47] S. Rodrguez. JPA implementations comparison: Hibernate, Toplink Essentials, OpenJPA and Eclipselink, Available At: newblock http://blog.eisele.net/2009/01/jpa-implementations-comparison.html 2011, Last Accessed Date: 20 June, 2017.

[48] A. Georges, D. Buytaert , L. Eeckhout. Statistically rigorous java performance evaluation. ACM SIGPLAN Notices, OOPSLA, Vol: 42, Issue-10, 2007, Pages: 57-76, Last Accessed Date: 20 June, 2017.

Appendix A

Confidence Interval

Evaluating the Confidence we compared the JPA providers based on the data analysis in Chapter 5 and for performing that we set up an experiment wherein we compared the overall performance of the implementation. The GC and CPU cycles over a range of heap sizes, cache size, and batch size were measured. We consider the various JPA benchmarks with numerous combinations of strategies as outlined in chapter 5, a range of cache sizes, batch size and heap sizes from the minimum to up to ten times the minimum with an increase of 1.25 increments with 15 heap sizes in total. We evaluated the confidence intervals for the statistically analysis based on the methodology state in [48], by using Visual VM to estimate 95% confidence for all the JPA API per JPA benchmark. By consequence, the confidence caclulated was then computed as is typically obtained from a pre-computed table. All runs are identical and independent of each other. The g independent results will be represented by $B1, B2, and B3, ..., Bg-1, Bg$.
The Mean

$$B = 1/g \sum_{i=1}^{g} B_i \tag{A.1}$$

However, the mean of independent running B mean provide us with single numerical value for the estimate of the expected value E$|B|$=SD. In order to know how good the estimation is provided by B mean for the empirical results, it is necessary to compute the variance;

$$V_b^2 \tag{A.2}$$

$$V_b^2 = 1/g - 1 \sum_{i}^{g} (Bi - \bar{B}) \tag{A.3}$$

A small variance indicates that the results are tightly clustered around B mean, and we can be confident that B mean is close to the E $|B|$, we can specify the interval of values that is highly likely to contain the true value of the parameter. We begin by specifying some high probability, say $(1 - \alpha)$. We then find the Lower bound L(B) and Upper Bound U(B) such that,

$$P[L(B) \leq \omega \leq U(B)] = (1 - \alpha). \tag{A.4}$$

The interval contains the true value of the parameters with probability, such as interval is $1 - \alpha * 100\%$ confidence interval.

$$LowerLimit = \bar{x} - E \tag{A.5}$$

$$UpperLimit = \bar{x} + E \tag{A.6}$$

Where:

$$E = z(\alpha/(2))\omega/\sqrt{n} \tag{A.7}$$

$$\alpha = 0.05$$

$$\omega = standarddeviation$$

$$\bar{x} = mean$$

$$n = Numberofobservations$$

$$E = Error$$

The confidence calculated means that 95% of the results fall within the interval. Through the thesis, the confidence interval is computed based on 30 independent runs.

Appendix B

Statistical Results

The statistics results include:

- Average Mean calculating GC cycle over JPA implementation using 30 execution per JPA Benchmarks per SQL query.
- Average Mean calculating CPU cycle over JPA implementation using 30 execution per JPA Benchmarks per SQL query.
- Standard Deviation of GC cycle over JPA implementation using 30 execution per JPA Benchmarks per SQL query.
- Standard Deviation of CPU cycle over JPA implementation using 30 execution per JPA Benchmarks per SQL query.

All the JPA benchmarks were incremented with every iteration and an analysis was performed to check whether that has increased or decreased the performance. Once the optimal size was obtained based on the average and standard deviation, we performed 30 execution per query per benchmark to derive the confidence of 95% in the results. The concept of overlapping/Nonoverlapping was simple we assumed that if the results shows high overlapping then it would be difficult to assess the difference in the JPA APIs based on that particular JPA benchmark. It was quite similar to overlapping/nonoverlapping concept stated in [48]. In order to solve the overlapping problem, we increased the sample size which would reduce the outlier/error rate, improve statistical the results and provide better confidence.

Table B.1: Average Mean calculating GC over JPA implementation using 30 execution per JPA Benchmarks per SQL query

Cache Size	Batch size	Heap Size	Hibernate	EclipseLink	OpenJPA	DataNucleus
500	500	120-125	2.24	3.85	1.64	11.68
1000	1000	125-130	2.38	4.03	1.62	10.49
1500	1500	131-135	3.39	3.57	0.44	12.42
2500	2500	136-140	3.02	3.80	1.19	14.19
3500	3500	141-145	2.52	3.86	0.28	11.14
4500	4500	146-150	2.49	3.89	1.36	16.70
5500	5500	151-155	3.67	3.96	1.28	11.35
6500	6500	156-160	2.96	4.68	1.03	13.01
7500	7500	166-170	2.65	3.52	1.01	10.76
8500	8500	171-175	3.84	3.28	0.55	13.12
9500	9500	176-180	2.89	3.64	0.43	12.68
10500	10500	181-185	4.29	3.74	0.41	12.85
11500	11500	186-190	2.69	4.40	0.63	10.78
12500	12500	191-195	2.83	4.19	0.66	12.79
13500	13500	196-200	2.58	4.17	0.52	13.21
14500	14500	201-205	2.96	2.94	1.21	12.06
15500	15500	206-210	2.26	4.04	0.60	13.76
16500	16500	211-215	2.93	4.08	0.46	9.96
17500	17500	216-220	3.24	3.75	0.36	13.21
18500	18500	221-225	2.84	3.85	0.85	12.06
19500	19500	226-230	3.04	4.49	0.92	13.76
20500	20500	231-235	2.96	3.65	0.87	9.96
25500	25500	236-240	2.65	3.33	0.80	12.18
30500	30500	241-245	3.84	3.89	0.68	12.34
35500	35500	246-250	2.89	4.44	0.68	12.58
40500	40500	251-255	4.29	4.98	0.63	13.52
45500	45500	256-260	2.69	4.76	1.54	11.63
50500	50500	261-265	2.83	4.21	0.98	15.72
55500	55500	266-270	2.58	4.89	0.73	10.94
60500	60500	271-275	2.96	4.67	1.78	12.10
		min	2.24	2.94	0.28	9.96
		max	4.29	4.98	1.78	16.70
		Q1	2.65	3.74	0.56	11.42
		Median	2.89	3.92	0.76	12.38
		Q3	3.035	4.3525	1.145	13.1875

Table B.2: Average Mean calculating CPU cycle over JPA implementation using 30 execution per JPA Benchmarks per SQL query

Cache Size	Batch size	Heap Size	Hibernate	EclipseLink	OpenJPA	DataNucleus
500	500	120-125	7.47	9.12	14.45	58.275
1000	1000	125-130	8.07	7.91	13.92	61.69
1500	1500	131-135	8.00	9.93	16.28	61.725
2500	2500	136-140	9.67	8.69	12.76	52.485
3500	3500	141-145	8.35	10.95	12.06	61.59
4500	4500	146-150	8.61	9.69	16.97	72.64
5500	5500	151-155	9.15	9.46	12.51	64.145
6500	6500	156-160	9.55	10.62	12.67	65.07
7500	7500	166-170	8.51	12.06	10.64	64.45
8500	8500	171-175	9.11	9.41	9.12	74.91
9500	9500	176-180	9.41	9.32	11.48	65.31
10500	10500	181-185	8.41	10.01	14.06	68.035
11500	11500	186-190	10.78	10.71	9.11	53.4
12500	12500	191-195	8.42	8.21	10.80	63.05
13500	13500	196-200	7.43	9.38	7.96	63.105
14500	14500	201-205	8.59	9.38	9.91	52.91
15500	15500	206-210	8.56	9.39	9.94	74.935
16500	16500	211-215	8.27	11.22	8.28	57.88
17500	17500	216-220	6.59	9.36	12.44	69.29
18500	18500	221-225	7.51	9.91	12.75	65.155
19500	19500	226-230	9.11	10.08	12.84	51.81
20500	20500	231-235	9.41	9.93	12.59	59.94
25500	25500	236-240	8.41	9.40	7.49	48.135
30500	30500	241-245	10.78	9.70	13.58	62.275
35500	35500	246-250	8.42	9.80	16.08	49.1
40500	40500	251-255	7.43	9.90	8.52	67.355
45500	45500	256-260	8.59	10.00	11.48	84.39
50500	50500	261-265	8.56	9.50	12.69	64.13
55500	55500	266-270	8.27	9.70	9.94	75.825
60500	60500	271-275	6.59	11.00	12.00	85.49
		min	6.59	7.91	7.49	48.14
		max	10.78	12.06	16.97	85.49
		Q1	8.12	9.38	9.94	58.69
		Median	8.46	9.70	12.25	63.62
		Q3	9.11	10.01	12.82	67.87

Table B.3: Standard Deviation of GC cycle over JPA implementation using 30 execution per JPA Benchmarks per SQL query

Cache size	Batch size	Heap size	Hibernate	EclipseLink	OpenJPA	DataNucleus
500	500	120-125	4.29	5.44	3.45	11.68
1000	1000	125-130	4.64	5.67	3.65	13.76
1500	1500	131-135	3.84	4.03	3.63	13.50
2500	2500	136-140	3.84	3.89	3.64	4.50
3500	3500	141-145	3.67	5.96	3.28	4.50
4500	4500	146-150	3.39	4.17	3.52	6.70
5500	5500	151-155	4.25	4.89	4.66	7.50
6500	6500	156-160	4.04	4.52	5.66	13.60
7500	7500	166-170	4.02	3.98	7.55	12.18
8500	8500	171-175	3.87	4.40	7.53	10.78
9500	9500	176-180	3.67	4.76	7.30	11.63
10500	10500	181-185	3.96	4.19	7.66	12.79
11500	11500	186-190	3.96	4.21	5.66	15.72
12500	12500	191-195	3.93	4.03	6.77	12.06
13500	13500	196-200	4.30	4.76	6.45	12.68
14500	14500	201-205	4.85	4.44	6.35	11.00
15500	15500	206-210	4.99	4.08	6.43	10.00
16500	16500	211-215	4.67	4.94	6.24	12.06
17500	17500	216-220	3.89	4.67	6.10	13.00
18500	18500	221-225	3.24	4.68	5.95	13.01
19500	19500	226-230	3.65	4.65	5.76	9.96
20500	20500	231-235	3.99	4.80	5.43	14.00
25500	25500	236-240	3.80	4.49	5.20	13.00
30500	30500	241-245	3.77	4.49	4.90	13.21
35500	35500	246-250	3.89	4.57	4.67	12.42
40500	40500	251-255	4.57	4.96	4.32	11.35
45500	45500	256-260	4.66	3.28	3.20	13.12
50500	50500	261-265	3.87	3.89	2.90	12.34
55500	55500	266-270	3.54	4.74	2.20	12.85
60500	60500	271-275	3.44	5.98	2.10	13.52
SD			0.43	0.60	1.63	2.64

Table B.4: Standard Deviation of CPU cycle over JPA implementation using 30 execution per JPA Benchmarks per SQL query

Cache size	Batch size	Heap Size	Hibernate	EclipseLink	OpenJPA	DataNucleus
500	500	120-125	17.47	19.12	14.45	58.28
1000	1000	125-130	18.07	27.91	13.92	61.69
1500	1500	131-135	18.00	29.83	16.28	61.73
2500	2500	136-140	19.67	28.98	32.00	52.49
3500	3500	141-145	18.35	20.95	32.00	61.59
4500	4500	146-150	18.61	29.96	31.00	72.64
5500	5500	151-155	19.15	29.94	30.00	64.15
6500	6500	156-160	19.55	29.92	23.00	65.07
7500	7500	166-170	18.51	30.50	24.00	64.45
8500	8500	171-175	19.11	39.41	21.00	74.91
9500	9500	176-180	19.41	39.41	22.48	65.31
10500	10500	181-185	18.41	30.01	24.90	68.04
11500	11500	186-190	20.78	32.71	29.11	53.40
12500	12500	191-195	18.47	38.21	20.80	63.05
13500	13500	196-200	17.43	39.81	23.60	63.11
14500	14500	201-205	18.59	29.81	29.91	52.91
15500	15500	206-210	18.56	29.38	44.00	74.94
16500	16500	211-215	18.27	33.22	44.56	57.88
17500	17500	216-220	16.59	29.81	44.65	69.29
18500	18500	221-225	17.51	29.91	32.75	65.16
19500	19500	226-230	19.11	30.41	32.84	51.81
20500	20500	231-235	19.41	29.93	32.59	59.94
25500	25500	236-240	18.41	29.40	37.49	48.14
30500	30500	241-245	20.78	29.70	43.50	62.28
35500	35500	246-250	18.43	29.80	36.80	49.10
40500	40500	251-255	17.43	29.90	38.52	67.36
45500	45500	256-260	18.59	30.00	33.48	84.39
50500	50500	261-265	18.56	29.50	32.69	64.13
55500	55500	266-270	18.27	29.70	25.00	75.83
60500	60500	271-275	16.59	33.00	32.00	85.49
SD			0.98	4.40	8.56	9.26

Appendix C

Query Results

The following Appendix contains Query results using local versus cloud database. This is in continuation from Section 4.2 explaining the query results through response time.

- Query 5 (SELECT COMMAND USING FULL OUTER JOIN + GROUP BY +ORDER BY+AGGREGATION)

  SELECT COUNT (E.ID) , D.ID FROM Employee E FULL OUTER JOIN Department D GROUP BY D.ID HAVING COUNT (E.ID)>0 ORDER BY D.ID HAVING D.ID<X

  The above query retrieves the count of Employee ID, Department ID within the Employee and Department table grouping Department ID and count of Employee ID >0 order Department ID in ascending order and Department ID<X. The task of this query is to pull out rows from the table. The average response time values in cloud and local databases are as shown in Table C.2, and by the value of X in Table C.1.

  These results show that the local database is performing well in Hibernate JPA implementation. At 30,000 entries the response time is two times higher with cloud, and at 60,000 entries it is almost 3.5 times higher. At 120,000 entries, the Cloud has almost 3.4 times higher response time, while at 240,000 entries the cloud has almost four times higher response times. At 480,000 entries cloud response time is almost 4.2 times higher.

  A similar response time was observed with DataNucleus JPA with the local database. At 30,000 entries the response time is nearly 6.1 times higher in cloud, and at 60,000 it is almost 9.9 times higher. At 120,000 entries, the cloud has

almost 8.1 times higher response time, while at 240,000 it is almost 9.2 times higher. At 480,000 entries the cloud response time is almost 11 times higher.
The error results in OpenJPA and EclipseLink JPA in Table C.1 were due to compatibility issues with FULL OUTER JOIN.

- Query 6(SELECT COMMAND USING LEFT OUTER JOIN)
  SELECT E.ID,E.salary,E.firstName,E.lastName, E.Salary, E.Date, D.branchName, FROM Employee E LEFT OUTER JOIN Department D ON E.ID=D.ID where E.Salary>0 and E.Salary<X
  The above query retrieves the Employee ID, Employee Name, Salary, Date and Branch Name within the Employee and Department table based on matching records and a salary range of 0 to X. The task of this query is to pull out large numbers of rows from the two tables. The average response time values in the cloud and traditional databases are as shown in Table C.4, and by the value of X in Table C.3.
  These results show that the local database is performing well with Hibernate JPA implementation. At 30,000 entries the response time is almost triple that of the cloud, and at 60,000 it is 3.4 times higher. At 120,000 entries, the Cloud has about 4.4 times higher response time, while at 240,000 it is 4.7 times higher. At 480,000 entries, the Cloud's response time is almost 5.9 times higher.
  In EclipseLink JPA the results of local database are also better compared to cloud. At 30,000 entries the response time is almost twice as long as in cloud, and at 60,000 it is 2.4 times higher. At 120,000 entries the cloud has almost 2.8 times higher response time, while at 240,000 entries it is close to 3.7 times higher. At 480,000 entries, the cloud response time is almost 4.4 times higher.
  Similar response times were observed with DataNucleus JPA with the local database. At 30,000 entries the response time is about 4.8 times with cloud, and at 60,000, it is almost 6.9 times higher. At 120,000 entries cloud has an 8.3 times higher response time, while at 240,000 entries, it is about 9.2 times higher. At 480,000 entries the cloud response time is almost 11.4 times higher.
  The error results in OpenJPA in Table C.4 are due to compatibility issues with LEFT OUTER JOIN.

- Query 7 (SELECT COMMAND USING LEFT OUTER JOIN + UNION ALL)
  SELECT E.ID,E.salary,E.firstName,E.lastName, E.Salary, E.Date FROM Em-

ployee E LEFT OUTER JOIN EmployeeProjects EP where E.Salary>0 and E.Salary<X
UNION ALL
SELECT E.ID,E.salary,E.firstName,E.lastName, E.Salary, E.Date FROM Employee E LEFT OUTER JOIN EmployeeProjects EP where E.Salary>0 and E.Salary>X
The above query retrieves the Employee ID, Employee Name, Salary and Date within the Employee and EmployeeProject table based on matching records and salary range 0 to X performing UNION ALL. The task of this query is to pull out large number of rows from the two tables. The average response time values in Cloud and local database are as shown in Table C.6, and values of X in Table C.5. These results show that the local Database performing well for this query in Hibernate JPA. At 30,000 entries the response time is three times higher in the cloud, and at 60,000 it is 3.8 times higher. At 120,000 entries the cloud has response time is 4.3 times higher, while at 240,000 entries it is 5.6 times higher. At 480,000 entries, the Cloud's response time is 8.9 times higher.
The results show that the local database is performing well in EclipseLink JPA implementation. At 30,000 entries the response time is almost 3.6 times that of the cloud, and at 60,000 entries it is 4.8 times higher. At 120,000 entries the cloud response time is almost five times higher, while at 240,000 it is close to 6.2 times higher. At 480,000 entries the cloud response time is almost ten times higher.
The error results in OpenJPA in Table C.6 are due to compatibility issues lack of a UNION ALL operator, while the DataNucleus JPA errors are due to lack of support for the SQL UNION ALL operator.

- Query 8 (SELECT COMMAND USING JOIN + NULL condition)
  SELECT E.ID,E.salary,E.firstName,E.lastName, E.Salary, E.Date FROM Employee E JOIN EmployeeProjects EP where EP.ID>0 and EP.ID<X and E.salary IS NULL
  The above query retrieves the Employee ID, Employee Name, Salary and Date within the Employee and EmployeeProject table based on matching records and salary range '0 'and 'X and Employee salary is NULL. The task of this query is to pull out large number of rows from the two tables. The average response time values in a cloud database and local database are shown in Table C.8, and the values of X in Table C.7.

These results show that the local Database is performing well in Hibernate JPA implementation. At 30,000 entries the response time is almost 3.6 times higher in cloud, and at 60,000 entries it is almost 4.2 times higher. At 120,000 entries the Cloud has almost 5.7 times higher response time, while at 240,000 entries it is almost 6.3 times higher. At 480,000 entries, the cloud response time is almost 8 times higher. The error results in EclipseLink JPA in Table C.8 are due to compatibility issues with IS NULL operator + JOIN which created an infinite loop.

In OpenJPA the results of the local database is better compared to the cloud. At 30,000 entries the response time is almost four times higher in cloud, and at 60,000 it is about 5.3 times higher. At 120,000 entries the cloud has almost six times higher response time, while at 240,000 it is almost 7.8 times higher. At 480,000 entries the cloud response time is almost ten times higher.

Similar response times were observed with DataNucleus JPA with local database. At 30,000 entries the response time is almost five times higher in cloud, and at 60,000 it is 6.6 times higher. At 120,000 entries the cloud response time is 7.8 times higher, while at 240,000 it is almost nine times higher. At 480,000 entries the cloud response time is almost 9.6 times higher.

- Query 9 (SELECT COMMAND USING RIGHT OUTER JOIN + GROUP BY +ORDER BY)

  SELECT SUM(E.salary) , EP.ProjectName FROM Employee E RIGHT OUTER JOIN EmployeeProjects EP ON E.PID=EP.PID and GROUP BY EP.ProjectName ORDER BY EP.ProjectName HAVING EP.ProjectName='% C%' AND EP.PID>'X' Above query retrieves the sum of Salary and Project Name from the Employee and EmployeeProject table based on left table matching records and group based on Project Name ordering records in ascending order and project name with 'C' in the middle of the string and project ID >'X' . The task of this query is to pull out large number of rows from the two tables. The average response time values in cloud database and local database are as shown in Table C.10, and values of X in Table C.9.

  These results show that the local database is performing well in Hibernate JPA implementation. At 30,000 entries the response time is almost 3.9 times higher in cloud, and at 60,000 it is about 4.7 times higher. At 120,000 entries the Cloud has an almost five times higher response time, while at 240,000 the it has close to a 6.1

times higher response time. At 480,000 entries the cloud response time is almost nine times higher. The error results in EclipseLink JPA and OpenJPA (Error) in Table C.10 are due to 'OutofMemory' Exception, as well as compatibility issues along with no support for RIGHT OUTER JOIN.
Similar response times were observed with DataNucleus JPA with the local database. At 30,000 entries the response time is almost 2.3 times that of Cloud, and at 60,000 it is about 4.3 times higher. At 120,000 entries the Cloud has almost 6.4 times higher response time, while at 240,000 it is almost seven times higher. At 480,000 entries the cloud response time is almost 8.2 times higher.

- Query 10(SELECT COMMAND USING JOIN)
  SELECT E.ID,E.salary,E.FirstName,E.LastName, E.Salary, E.Date, D.BranchName, FROM Employee E LEFT OUTER JOIN Department D ON E.ID=D.ID where E.Salary>0 and E.Salary<'X' and E.Date<DATEADD(day, DATEDIFF(day, 0, GETDATE()), −1)
  The above query retrieves the sum of Salary and Project Name within the Employee and EmployeeProject table, based on left table matching records and group based on Project Name ordering records in ascending order and project name with, C in the middle of the string,project ID >X and Date less than the current date. The task of this query is to pull out large number of rows from the two tables. The average response time values in a cloud and local database are as shown in Table C.12, and values of X in Table C.11.
  These results also show that the local database is performing well in Hibernate JPA implementation. At 30,000 entries the response time is almost five times that of with cloud, and at 60,000 it is almost 6.8 times higher. At 120,000 entries the cloud has about ten times higher response time, while at 240,000 it is about 11.4 times higher. At 480,000 entries, the cloud response time is almost 12 times higher.
  The error results in EclipseLink JPA, OpenJPA and DataNucleus JPA in Table C.12 are due to compatibility issues with DATEADD, DATEDIFF and GETDATE function operator.

- Query 11(SELECT COMMAND USING JOIN + IN OPERATOR)

  SELECT E.ID,E.salary,E.firstName,E.lastName, E.Salary, E.Date, D.BranchName, FROM Employee E JOIN Department D WHERE E.ID IN (1, 30000) AND D.ID

>0 AND D.ID <X

The above query retrieves the Employee Name, Salary , Date and Branch Name within the Employee and Department table, based on Join matching records and extracting Employee ID between (1, 30000) using an IN operator and Department ID in range 1 to X. The task of this query is to pull out large number of rows from the two tables. The average response time values in cloud database and traditional databases are as shown below in Table C.14, and values of X in Table C.13.

These results show that the local Database is performing well in Hibernate JPA implementation. At 30,000 entries the response time is almost twice that with cloud, and at 60,000 it is almost 3.8 times higher. At 120,000 entries the cloud has almost 4.6 times higher response time, while at 240,000 entries it is six times higher. At 480,000 entries, the cloud response time is almost 6.2 times higher.

The results also show that the local database is performing marginally well in EclipseLink JPA implementation. At 30,000 entries the response time is almost three times higher in cloud,and at 60,000 it is also three times higher. At 120,000 entries the Cloud has almost five times higher response time, while at 240,000 it is almost 5.2 times. At 480,000 entries the cloud response time is almost six times higher.

The error results in Open JPA(-) in Table C.14 are due to compatibility issues with the IN operator.

Similar response times were observed with DataNucleus JPA with the local database. At 30,000 entries the response time is almost five times higher in cloud, and at 60,000 it is almost 6.3 times higher. At 120,000 entries the Cloud has almost 8.4 times higher response time, while at 240,000 it is almost 9.7 times higher. At 480,000 entries, the cloud response time is close to 10.1 times higher.

- Query 12 (SELECT COMMAND USING JOIN + GROUP BY + AGGREGATION)

  SELECT COUNT(E.ID), D.ID, E.Salary from Employee E JOIN Department D GROUP BY D.ID, E.Salary HAVING (COUNT(E.ID)> 0 OR E.Salary>0 and E.Salary<' X')

  Above query retrieves the total Employee ID, Salary and groups them using Department ID and salary within the Employee and Department table, with Employee ID count >0 or Employee salary range 1 to X. The task of this query is to pull out large number of rows from the two tables. The average response time

values in cloud and local databases are as shown in Table C.16, and the values of X in Table C.15.

These results show that the local database is again performing well in Hibernate JPA implementation. At 30,000 entries the response time is almost 3 times higher in cloud, and at 60,000 it is 3.5 times higher. At 120,000 entries the cloud response time is close to 4.1 times higher, while at 240,000 it is 5 times higher. At 480,000 entries the cloud response time is almost 5.2 times higher.

The results show that the local database is performing marginally well in EclipseLink JPA implementation. At 30,000 entries the response time is almost 1.2 times higher than with cloud, and at. 60,000 it is 2.4 times higher. At 120,000 entries the Cloud response time is close to 4.7 times higher, while at 240,000 entries it is almost 5.3 times higher. At 480,000 entries the cloud response time is about 7.9 times higher.

With OpenJPA,the results with 30,000 entries show the cloud response time is almost five times higher in cloud, and nearly 5.7 times higher at 60,000. At 120,000 entries the Cloud has almost 6.4 times higher response time, while at 240,000 it is almost 7.2 times higher. At 480,000 entries the cloud response time is almost 8.2 times higher.

Similar response times were observed with DataNucleus JPA and the local database. At 30,000 entries the response time is almost six times higher in cloud, and at 60,000 nearly 7.4 times higher. At 120,000 entries the cloud has almost 7.2 times higher response time, while at 240,000 entries it is almost 8.3 times higher. At 480,000 entries the cloud response time is close to 9.1 times higher.

- Query 13 (SELECT COMMAND USING JOIN+AGGREGATION)
  SELECT E.ID, D.ID, E.Salary, SYSDATE, $StringLiteral, 42 * 37$ Expression from Employee E JOIN Department D ON E.DID= D.ID and E.Salary>0 and E.Salary<'X'

  The above query retrieves the Employee ID, Department ID, salary and System current date within the Employee and Department table with Employee Salary in the range of 1 to X. The task of this query is to pull out large number of rows from the two tables. The average response time values in the cloud and traditional databases are as shown in Table C.18, and the values of X in Table C.17.

  These results show that the local Database is performing well in Hibernate JPA implementation. At 30,000 entries the response time is almost double in cloud,

and at 60,000 it is three times higher. At 120,000 entries the cloud has almost 3.7 times higher response time, while at 240,000 entries it is about 4.8 times higher. At 480,000 entries the cloud response time is almost six times higher.

The results also show that the local database is performing marginally well in EclipseLink JPA implementation. At 30,000 entries the response time is almost double in cloud, and at 60,000 it is 3.6 times higher. At 120,000 entries the cloud response time is nearly 5.3 times higher, while at 240,000 it is almost 3.6 times higher. At 480,000 entries the cloud response time is close to 4.6 times higher.

The response times for OpenJPA at 30,000 entries is almost 2.4 times higher in Cloud,and at 60,000 almost 7.4 times higher. At 120,000 entries the Cloud has almost 3.7 times higher response times, while at 240,000 it is almost 4.5 times higher. At 480,000 entries the cloud response time is nearly 4.2 times higher.

Similar response times were observed with DataNucleus JPA with the local database. At 30,000 entries the response time is almost 4.5 times higher in cloud, and at 60,000 it is about 4.4 times higher. At 120,000 entries the cloud has nearly 4.6 times higher response time, while at 240,000 it is close to 5.3 times higher. At 480,000 entries the cloud response time is almost 5.8 times higher.

- Query 14(SELECT COMMAND USING SUBQUERY)
  SELECT E.ID,E.firstName,E.lastName, E.Salary, E.Date FROM Employee E WHERE E.ID <> 1 AND E.PID = (SELECT EP.PID FROM EmployeeProject EP WHERE EP.ProjectName='M%') and E.ID<X

  Above query retrieves the ID, Employee Name, Employee, Employee salary and date from Employee, where Employee ID <>1 has an Employee Project Name range between M% and Employee ID <X. The average response time values in cloud and local databases are shown in Table C.20, and values of X in Table C.19. These results show that the local database performs well with Hibernate JPA implementation. At 30,000 entries the response time is almost 2.5 times higher than with cloud, and at 60,000 it is three times higher. At 120,000 entries the cloud response time is nearly 2.7 times higher, while at 240,000 it is close to 3.6 times higher. At 480,000 entries the cloud response time is almost 4.1 times higher.

  The results also show that the local database is performing marginally well in EclipseLink JPA implementation. At 30,000 entries the cloud response time is almost 3.5 higher than with the local database, and at 60,000 it is six times higher. At 120,000 entries the cloud response is nearly 6.8 times higher, while at 240,000

entries it is nearly 8.3 times higher. At 480,000 entries the cloud response time is almost 9.2 times higher. In OpenJPA the errors are due to compatibility issues with sub-queries and wild card characters '%'.
In DataNucleus JPA, the response time at 30,000 entries is almost four times higher in cloud, and at 60,000 it is close to 4.3 times higher. At 120,000 entries the cloud response is nearly 5.6 times higher, while at 240,000 it is almost 7.2 times higher. At 480,000 entries the cloud response time is 8.7 times higher.

- Query 15 (SELECT COMMAND USING JOIN+ SUBQUERY)
  SELECT .ID,E.salary,E.firstName,E.lastName, E.Salary, E.Date FROM Employee E JOIN Department D JOIN EmployeeProject EP ON E.ID=D.ID WHERE E.ID BETWEEN (1,400000) and EP.EPID >0 and EP.EPID <X and E.salary> 10000
  The above query retrieves the Employee ID, Department ID, salary and Date within the Employee and Department table with Employee Salary in the range of greater than 10000 and E.PID in range of '1' to X. The task of this query is to pull out large number of rows from the two tables. The average response time values in cloud and local databases are as shown in Table C.22, and values of X in Table C.21.
  These results show that the local database is performing well in Hibernate JPA implementation. At 30,000 entries the response time is almost 2.6 times higher with cloud, and at 60,000 it is about 3.2 times higher. At 120,000 entries the cloud has nearly 3.5 times higher response time, while at 240,000 it is close to 4.8 times higher. At 480,000 entries the cloud response time is almost six times higher.
  The results show that the local database performs marginally well with EclipseLink JPA implementation. At 30,000 entries the response time is almost 4.3 times higher in cloud, and at 60,000 it is 5.9 times higher. At 120,000 entries the cloud has nearly 6.3 times higher response time, while at 240,000 entries it is close to 7.4 times higher. At 480,000 entries the cloud response time is almost 7.7 times higher.
  With OpenJPA, at 30,000 entries the response time is almost 3.4 times in cloud, and at 60,000 it is nearly 4.4 times higher. At 120,000 entries, the cloud response time is close to 5.8 times higher, while at 240,000, it is almost 6.5 times higher. At 480,000 entries the cloud response time is about 8.2 times higher.
  These results show that the local Database is performing well in DataNucleus JPA implementation. At 30,000 entries the response time is almost 2.6 times higher with cloud, and at 60,000 it is nearly 3.2 times higher. At 120,000 entries the cloud

response time is close to 3.5 times higher, while at 240,000 it is almost 4.8 times higher. At 480,000 entries the cloud response time is almost six times higher.

- Query 16 (SELECT COMMAND USING JOIN+ WILD CARD CHARACTER)
  SELECT E.ID,E.Salary,E.firstName,E.lastName FROM Employee E JOIN Department D ON E.ID=D.ID WHERE E. Name = 'b%' and E.Salary NOT BETWEEN $(10000, 500000) and D.ID > 0 and D.ID <$X.
  The above query retrieves the Employee ID, Employee Name and salary within the Employee and Department table with E. Name = 'b% and Employee Salary in Not BETWEEN (10000, 500000)and Department ID range of 1 to X. The task of this query is to pull out large number of rows from the two tables. The average response time values in cloud and local databases are as shown in Table C.24, and the values of X in Table C.23.
  These results show that the local Database is performing well in Hibernate JPA implementation. At 30,000 entries the cloud response time is almost 2.6 times higher, and 60,000 it is nearly three times higher. At 120,000 entries the cloud has close to 4.2 times higher response time, while at 240,000 it is about five times higher. At 480,000 entries the cloud response time is almost 7.8 times higher.
  The results also show that the local database performs marginally well in EclipseLink JPA implementation. At 30,000 entries the response time is almost 2.5 times higher in Cloud, and at 60,000 it is 4.7 times higher.
  At 120,000 entries the cloud has nearly 8.2 times higher response time, while at 240,000 it is about 9.6 times. At 480,000 entries, the Cloud's response time is close to 10.1 times higher.
  These results show that the local database is performing well with OpenJPA implementation. At 30,000 entries the response time is almost 3.6 times higher than cloud, and at 60,000 it is nearly 4.5 times higher. At 120,000 entries, the cloud has close to 6.9 times higher response times, while at 240,000 it is almost eight times. At 480,000 entries the cloud response time is close to 11.1 times higher. With DataNucleus JPA, at 30,000 entries the response time is almost 3.5 times higher with cloud, and at 60,000 it is about 4.5 times higher. At 120,000 entries the cloud has nearly 4.9 times higher response time, while at 240,000 it is close to 5.7 times higher. At 480,000 entries the Clouds response time is almost 6.2 times higher.

Table C.1 Data entries of the Query 5

	30,000 entries	60,000 entries	120,000 entries	240,000 entries	480,000 entries
X	5,000	33,000	100,000	220,000	320,000

Table C.2 Response Time Values of different Entries for Local and Cloud Database in Query5

Response time	Retrieved results	Hibernate		EclipseLink		OpenJPA		DataNucleus	
		Local Database (ms)	Cloud Database (ms)	Local Database (ms)	Cloud Database (ms)	Local Database (ms)	Cloud Database (ms)	Local Database (ms)	Cloud Database (ms)
30,000 entries	1,022	332	664	Error	Error	Error	Error	306	1,836
60,000 entries	2,336	838	2,944	Error	Error	Error	Error	1,026	10,260
120,000 entries	5,635	1,065	3,621	Error	Error	Error	Error	1,990	15,920
240,000 entries	10,553	1,264	5,056	Error	Error	Error	Error	3,340	30,728
480,000 entries	17,589	2,453	10302	Error	Error	Error	Error	7,083	78,621

Table C.3 Data Entries of Query 6

	30,000 entries	60,000 entries	120,000 entries	240,000 entries	480,000 entries
X	12,000	45,000	11,550	23,500	42,000

Table C.4 Response Time Values of Different Entries for Local and Cloud Databases in Query 6

Response time	Retrieved results	Hibernate JPA		EclipseLink JPA		OpenJPA		DataNucleus	
		Local Database (ms)	Cloud Database (ms)	Local Database (ms)	Cloud Database (ms)	Local Database (ms)	Cloud Database (ms)	Local Database (ms)	Cloud Database (ms)
30,000entries	3,540	455	1,365	667	1,334	Error	Error	788	3,788
60,000entries	14,755	638	2,164	944	2,265	Error	Error	1,022	7,053
120,000entries	13,949	1,035	4,554	1,421	3,978	Error	Error	2,183	18,119
240,000entries	18,692	2,172	10207	2,526	9,346	Error	Error	3,522	32,408
480,000entries	37,379	4,325	25517	2,253	9,916	Error	Error	4,322	49,264

Table C.5 Data entries of the Query 7

	30,000 entries	60,000 entries	120,000 entries	240,000 entries	480,000 entries
X	25,000	50,000	10,0000	200,000	400,000

Table C.6 Response Time Values of Different Entries for Local and Cloud Database in Query 7

Response time	Retrieved results	Hibernate		EclipseLink		OpenJPA		DataNucleus	
		Local Database (ms)	Cloud Database (ms)	Local Database (ms)	Cloud Database (ms)	Local Database (ms)	Cloud Database (ms)	Local Database (ms)	Cloud Database (ms)
30,000 entries	4442	204	612	324	1,373	Error	Error	Error	Error
60,000 entries	6,645	1,097	4,162	882	4,335	Error	Error	Error	Error
120,000 entries	7,756	1,590	6,857	1,162	5,621	Error	Error	Error	Error
240,000 entries	10,273	3,080	17342	1,690	10,478	Error	Error	Error	Error
480,000 entries	27,463	7,083	63068	2,343	23,002	Error	Error	Error	Error

Table C.7 Data Entries of Query 8

	30,000 entries	60,000 entries	120,000 entries	240,000 entries	480,000 entries
X	15,000	30,000	45,000	60,000	75,000

Table C.8 Response Time Values of Different Entries for Local and Cloud Database in Query 8

Response time	Retrieved results	Hibernate		EclipseLink		OpenJPA		DataNucleus	
		Local Database (ms)	Cloud Database (ms)	Local Database (ms)	Cloud Database (ms)	Local Database (ms)	Cloud Database (ms)	Local Database (ms)	Cloud Database (ms)
30,000 entries	5,228	486	1745	Error	Error	424	1,696	574	2870
60,000 entries	8,677	763	3,244	Error	Error	688	3,674	890	5,874
120,000 entries	19,732	1,947	11093	Error	Error	1,362	8,172	1,203	9,354
240,000 entries	37,362	2,631	16526	Error	Error	1,858	14,496	2,441	19,038
480,000 entries	45,422	4,473	34967	Error	Error	2,756	27,560	4,823	46,302

Table C.9 Data Entries of Query 9

	30,000 entries	60,000 entries	120,000 entries	240,000 entries	480,000 entries
X	15,000	40,000	100,000	125,000	450,000

Table C.10 Response Time Values of Different Entries for Local and Cloud Database in Query 9

Response time	Retrieved results	Hibernate		EclipseLink		OpenJPA		DataNucleus	
		Local Database (ms)	Cloud Database (ms)	Local Database (ms)	Cloud Database (ms)	Local Database (ms)	Cloud Database (ms)	Local Database (ms)	Cloud Database (ms)
30,000 entries	354	563	2,197	Error	Error	Error	Error	274	632
60,000 entries	1,455	836	3,929	Error	Error	Error	Error	553	2,385
120,000 entries	5,399	1,103	5,526	Error	Error	Error	Error	1,733	11,037
240,000 entries	18,692	1,441	17452	Error	Error	Error	Error	2,873	20,112
480,000 entries	27,377	2,823	25492	Error	Error	Error	Error	4,743	38,869

Table C.11 Data entries of the Query 10

	30,000 entries	60,000 entries	120,000 entries	240,000 entries	480,000 entries
X	750	1,500	11,000	25,000	45,000

Table C.12 Response Time Values of Different Entries for Local and Cloud Database in Query 10

Response time	Retrieved results	Hibernate		EclipseLink		OpenJPA		DataNucleus	
		Local Database (ms)	Cloud Database (ms)	Local Database (ms)	Cloud Database (ms)	Local Database (ms)	Cloud Database (ms)	Local Database (ms)	Cloud Database (ms)
30,000 entries	354	442	2210	Error	Error	Error	Error	Error	Error
60,000 entries	1,455	786	5,342	Error	Error	Error	Error	Error	Error
120,000 entries	5,399	1,285	12850	Error	Error	Error	Error	Error	Error
240,000 entries	18,692	2,746	31340	Error	Error	Error	Error	Error	Error
480,000 entries	27,377	4,583	54996	Error	Error	Error	Error	Error	Error

Table C.13 Data Entries of Query 11

	30,000 entries	60,000 entries	120,000 entries	240,000 entries	480,000 entries
X	10,000	20,000	40,000	120,000	140,000

Table C.14 Response Time Values of Different Entries for Local and Cloud Database in Query 11

Response time	Retrieved results	Hibernate		EclipseLink		OpenJPA		DataNucleus	
		Local Database (ms)	Cloud Database (ms)	Local Database (ms)	Cloud Database (ms)	Local Database (ms)	Cloud Database (ms)	Local Database (ms)	Cloud Database (ms)
30,000 entries	5,332	345	1,035	422	1,266	-	-	656	3,280
60,000 entries	12,433	586	2,235	677	2,031	-	-	973	6,130
120,000 entries	22,838	756	3,777	1,033	5,164	-	-	1,143	9,602
240,000 entries	32,884	1,044	5,427	1,393	7,244	-	-	2,386	23,145
480,000 entries	68,848	2,452	15203	2,343	14,563	-	-	4,394	43,500

Table C.15 Data Entries of Query 12

	30,000 entries	60,000 entries	120,000 entries	240,000 entries	480,000 entries
X	15,000	45,000	100,000	150,000	145,000

Table C.16 Response Time Values of Different Entries for Local and Cloud Database in Query 12

Response time	Retrieved results	Hibernate		EclipseLink		OpenJPA		DataNucleus	
		Local Database (ms)	Cloud Database (ms)	Local Database (ms)	Cloud Database (ms)	Local Database (ms)	Cloud Database (ms)	Local Database (ms)	Cloud Database (ms)
30,000 entries	4,384	339	1,017	426	512	625	3,125	475	2,850
60,000 entries	6,455	583	2,037	673	1,616	753	4,230	697	5,157
120,000 entries	13,929	763	3,128	936	4,379	976	6,247	864	6,220
240,000 entries	29,561	1,036	5,179	1,374	7,283	1,087	7,827	1,566	12,998
480,000 entries	42,422	1,445	7,514	2,173	17,169	1,975	16,195	2,449	22,286

Table C.17 Data Entries of Query 13

	30,000 entries	60,000 entries	120,000 entries	240,000 entries	480,000 entries
X	750	1,500	11,000	25,000	45,000

Table C.18 Response Time Values of Different Entries for Local and Cloud Databases in Query 13

Response time	Retrieved results	Hibernate		EclipseLink		OpenJPA		DataNucleus	
		Local Database (ms)	Cloud Database (ms)	Local Database (ms)	Cloud Database (ms)	Local Database (ms)	Cloud Database (ms)	Local Database (ms)	Cloud Database (ms)
30,000 entries	578	245	490	374	1,122	578	1388	523	2,356
60,000 entries	1,052	563	1,689	684	2,464	772	5714	624	2,747
120,000 entries	5,384	783	2,898	973	5,157	873	3,235	1,023	4,706
240,000 entries	12,664	1,057	5,075	1032	3,717	1,007	4,534	1,532	8,110
480,000 entries	24,374	1,852	11125	2,653	12,204	1,974	8,290	3,753	21,768

Table C.19 Data Entries of Query 14

	30,000 entries	60,000 entries	120,000 entries	240,000 entries	480,000 entries
X	10,000	20,000	40,000	80,000	160,000

Table C.20 Response Time Values of Different Entries for Local and Cloud Database in Query 14

Response time	Retrieved results	Hibernate		EclipseLink		OpenJPA		DataNucleus	
		Local Database (ms)	Cloud Database (ms)	Local Database (ms)	Cloud Database (ms)	Local Database (ms)	Cloud Database (ms)	Local Database (ms)	Cloud Database (ms)
30,000 entries	6,563	249	623	362	1,267	Error	Error	548	2,192
60,000 entries	33,649	578	1,733	466	2,796	Error	Error	838	4,693
120,000 entries	64,746	1,383	3,735	825	5,610	Error	Error	1,847	10,345
240,000 entries	144,789	2,142	7,712	2,473	19,784	Error	Error	1,938	13,455
480,000 entries	274,927	2,786	11424	6,322	58,164	Error	Error	2,717	23,638

Table C.21 Data Entries of Query 15

	30,000 entries	60,000 entries	120,000 entries	240,000 entries	480,000 entries
X	25,000	50,000	100,000	20,0000	400,000

Table C.22 Response Time Values of Different Entries for Local and Cloud Databases in Query 15

Response time	Retrieved results	Hibernate		EclipseLink		OpenJPA		DataNucleus	
		Local Database (ms)	Cloud Database (ms)	Local Database (ms)	Cloud Database (ms)	Local Database (ms)	Cloud Database (ms)	Local Database (ms)	Cloud Database (ms)
30,000 entries	4,067	357	930	393	984	424	1,527	442	1547
60,000 entries	6,774	496	1,488	593	2,789	757	3,407	695	3,128
120,000 entries	10,736	678	2,848	795	6,519	846	5,838	1,044	5,116
240,000 entries	47,436	1,353	6,765	1,034	9,927	1,979	15,832	1,404	8,003
480,000 entries	147,433	1,863	14535	1,535	15,403	2,049	22,744	2,849	17,664

Table C.23 Data entries of the Query 16

	30,000 entries	60,000 entries	120,000 entries	240,000 entries	480,000 entries
X	17,500	25,000	100,000	155,000	455,000

Table C.24 Response Time Values of different entries for Local and Cloud Database in Query 16

Response time	Retrieved results	Hibernate		EclipseLink		OpenJPA		DataNucleus	
		Local Database (ms)	Cloud Database (ms)	Local Database (ms)	Cloud Database (ms)	Local Database (ms)	Cloud Database (ms)	Local Database (ms)	Cloud Database (ms)
30,000 entries	1,037	255	663	342	1,472	284	966	355	923
60,000 entries	2,636	473	1,515	425	2,508	446	1,964	524	1,677
120,000 entries	4,736	863	3,024	746	4,701	633	3,672	1,284	4,494
240,000 entries	18,336	1,253	6,015	1,342	9,931	1,342	8,723	2,243	10,778
480,000 entries	38,383	2,041	12246	2,123	16,349	1,453	1,1915	2,674	16,044

Appendix D

Results of JPA Implementations using Number of classes and Thread Counts (Live/Daemon) in Local versus Cloud Database

The following Appendix contains all four JPA implementations results using the local versus cloud database. This is in continuation from Section 4.1 (Chapter 4) verifying the query results through parameters such as Number of classes called and Number of Live/Daemon Threads.

D.1 Analysis of JPA Framework

D.1.1 Hibernate JPA Local versus Cloud Database

- Number of Classes Loaded : The number of classes and objects loaded while executing a complicated query can be measured, and used to analyze the correlation between the number of classes called over shared or local Metaspace. The visual description of the number of classes loaded and the number of classes shared in a session while executing a query in an application was scaled in the range of 0 to 6000, based on the number of objects assigned. Figure D.1 shows classes in JPA, with the number of classes loaded to execute each query and using fewer classes in a cloud than a physical database. A constant pattern of low number of the total

of loaded classes of 2160 (Q3), and as high as 2250 (Q16), formed the benchmarks for the cloud. Figure D.1 indicates a constant pattern for both Q1 and Q16, with the low-class rate expending most of the execution time.

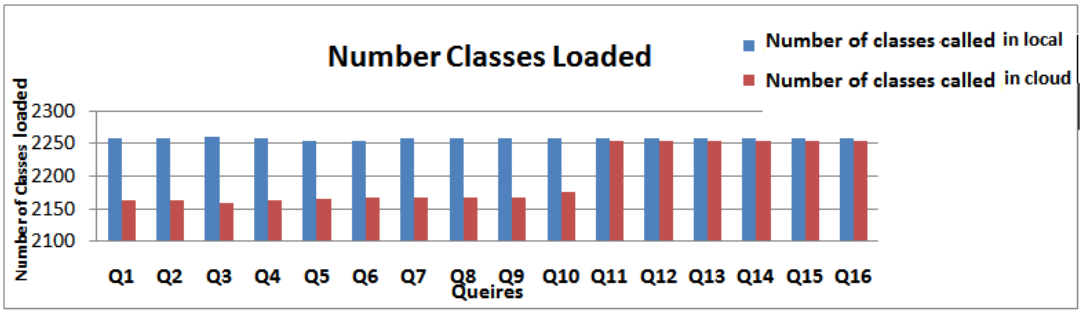

Figure D.1: Number of classes called in Hibernate JPA local versus cloud database

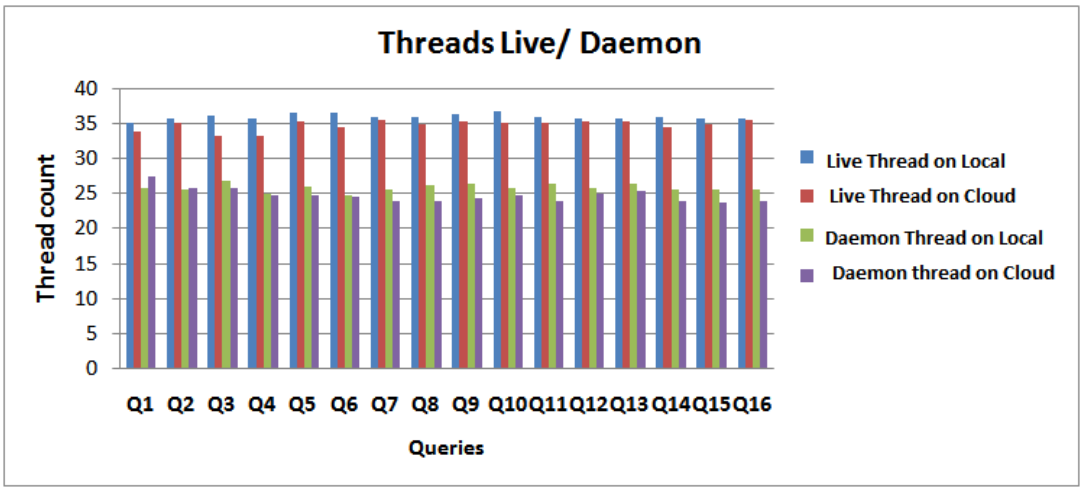

Figure D.2: Number of Thread (Live/Daemon) in Hibernate JPA local versus cloud database

- Threading (Live / Daemon): The analysis allowed us to identify and trace how many live and daemon threads are required to execute a specific query, and the behavior of the thread while executing through a local database or a cloud database. Figure D.2 shows the average mean of per query, with the number of Live/Daemon threads in Hibernate JPA API. There is a periodically high number of Live thread rates (issued per query in a thread model) of approximately 45 and 37 in both physical and cloud database (Q10 and Q16) for live threads, and a medium rate of approximately 15 to 28 threads per query in the daemon. In contrast, the query

dispersal rate of the cloud database was low compared to the local database, as shown in Figure D.2. Although the number of threads (live) was high in the local database compared to the cloud with an exception in daemon showing high rate in the cloud database. The range of thread was between 0 - 40 reaching as high as 35 on an average for both local and cloud database for live threads.

D.1.2 EclipseLink JPA Local versus Cloud Database

- Number of Classes : To determine specific object behavior, the access from class constructor methods which are invoked after object allocation are necessary to understand the performance of the JPA implementation. Figure D.3 shows the difference between several classes loaded in an SQL operation. Overall, the graph indicates that the local database performed quite well compared to the cloud database, but in Q15 the class size performance degraded to more than 2500 classes loaded in one operation, which adversely impacted the performance of the implementation. Some queries are not shown in the graph, because of errors and incompatibility issues with the EclipseLink JPA.

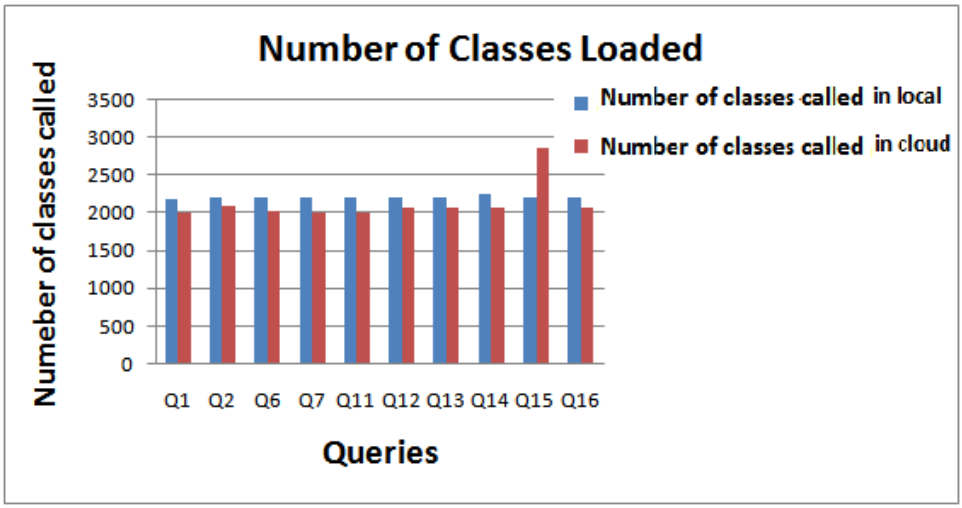

Figure D.3: Number of classes called in EclipseLink JPA local versus cloud database

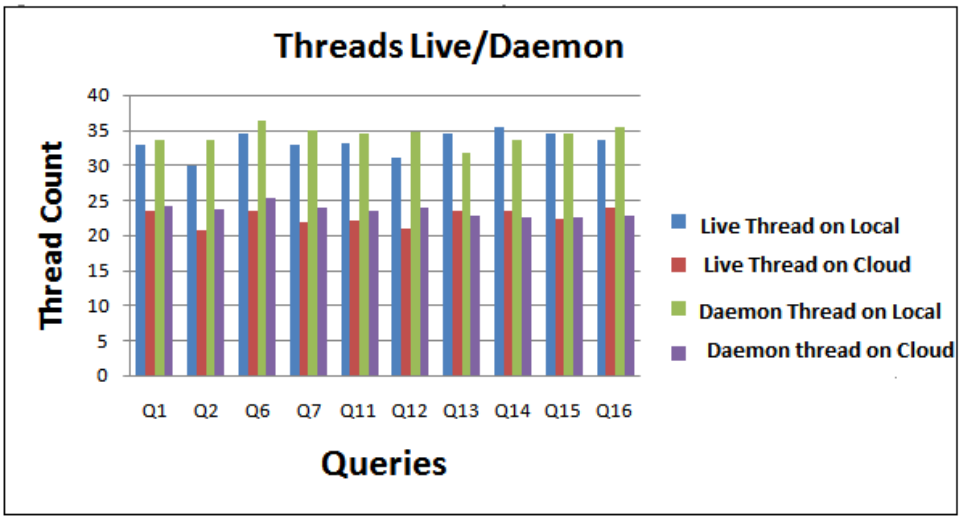

Figure D.4: Number of Thread (Live/Daemon) in EclipseLink JPA local versus cloud database

- Live and Daemon Threads: Due to the abstract execution environment implemented by the ORM platforms delivering EclipseLink JPA and the Entity Model in an object context, the entities are persisted in a private cache as one unit of work or thread, performing object instantiate for each entity id persisted to the database. Figure D.4 shows that the local database performed well compared to the cloud database. Although the number of Live threads in the local database is as high as 35, a number of live threads in the cloud was up to 37. The results in both databases were similar when handling Daemon thread. In Q6 and Q16, the cloud database was significantly outperformed by the local database, although the local database consumed more live threads in Q14. The results show that both persistence platforms are unable to manage threads effectively, and require a better approach. A more detailed analysis using query response time is discussed in the Appendix C which further elaborates the results.

D.1.3 OpenJPA Local versus Cloud database

- Number of classes Loaded: The OpenJPA Enhancer optimizes the run-time performance of an application, with flexible lazy loading and dynamic, fast dirty tracking, as discussed in the previous chapters. Figure D.5 shows a comparative analysis of an OpenJPA implementation using local versus cloud databases, and the performance of the local database is relatively good. With the local database,

the number of classes required for execution of the operation was locally cached, which further improved the results. A constant number of classes almost three times more than the local database were loaded into the cloud database, and the results were as high as 2,300 bytes for the number of shared and non-shared classes in an execution, which was about 250 bytes more than the local database. Overall the performance showed a mixed results with the number of classes loaded using local versus cloud database.

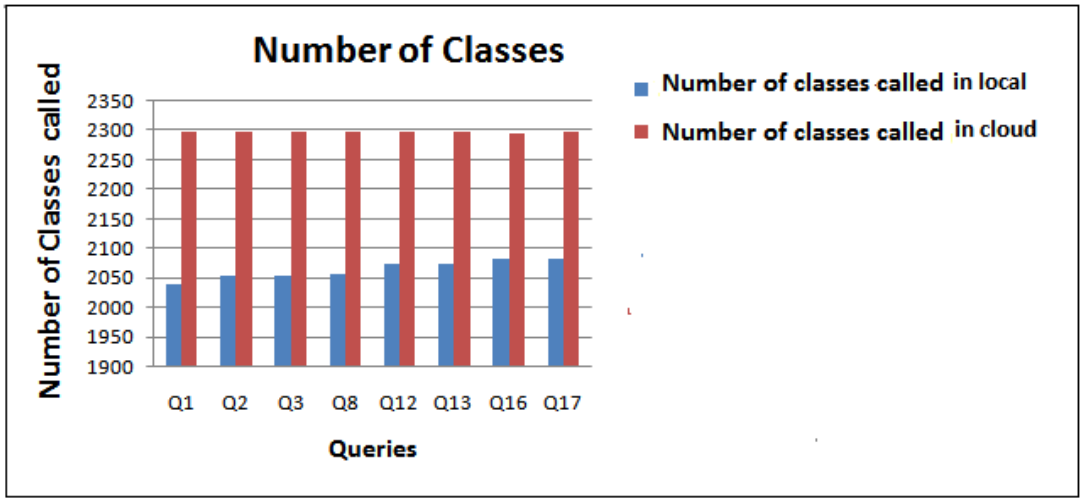

Figure D.5: Number of classes called in OpenJPA local versus cloud database

- Thread scheduling (Live/ Daemon): OpenJPA performed well with the local database with live and daemon threads both compared to the cloud database. Figure D.6 is a graph of all the queries that were executed in OpenJPA with both local and cloud databases. The performance of Q12 was significantly worse than other queries in the cloud database, reaching almost 320 live threads before becoming non-responsive, while other queries in cloud database were between the range of 0 and 45. The results of Q1, Q2, Q3, and Q8 showed that the local and cloud databases used the same number of live threads, and a similar pattern was observed in Q3, Q8, and Q13. Overall, the performance of local and cloud databases was similarly based on the number of threads. As the thread is a non-conforming standard to measure the performance of a JPA API, testing the validity of the results might require consideration of more hot point for the data set.

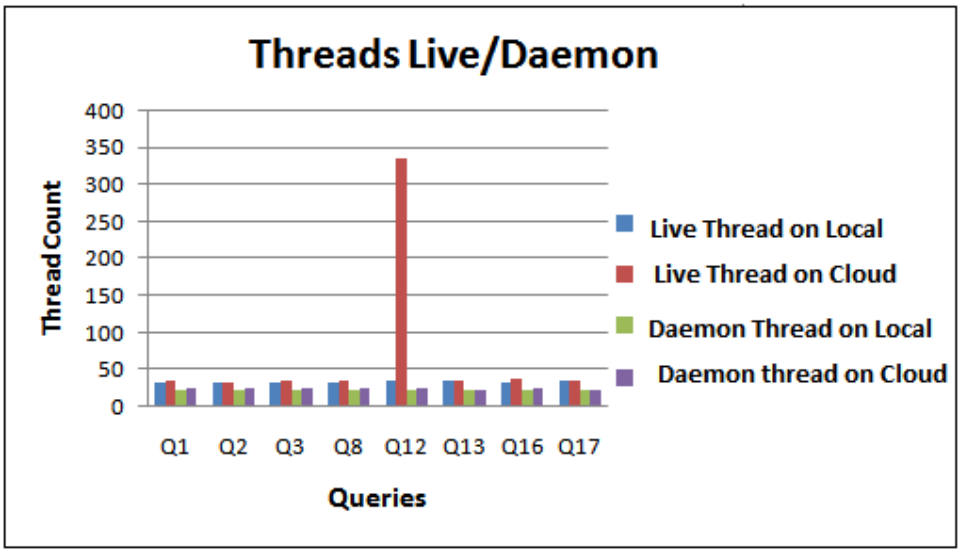

Figure D.6: Number of classes in OpenJPA local versus cloud database

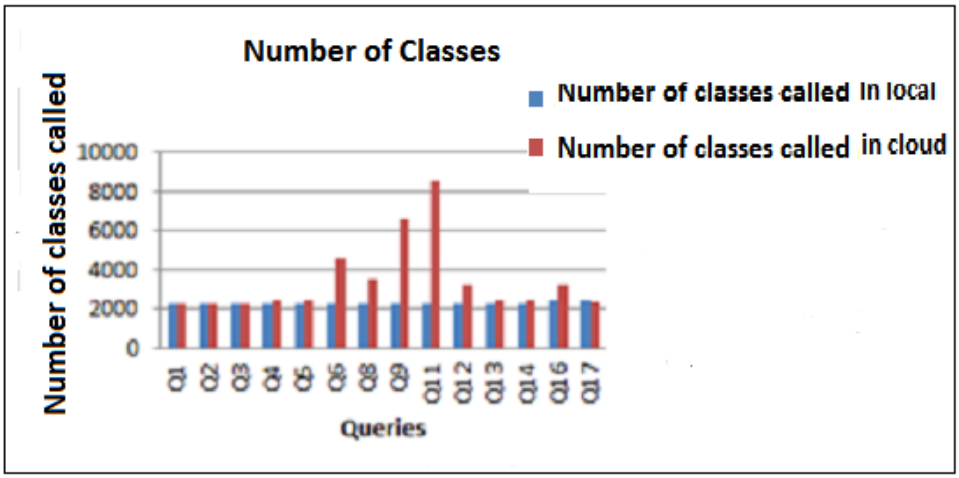

Figure D.7: Number of classes called in DataNucleus JPA local versus cloud database

D.1.4 DataNucleus JPA Local versus Cloud database

- Threading (Live/ Daemon):In DataNucleus, JPA Live threads performed better with the local database than with the cloud database, while Daemon threads were almost identical for both platforms. Figure D.7 shows the number of threads used while persisting data on a local and cloud database, using DataNucleus JPA. The performance of Q1, Q3 and Q17 (an altered Q15) was almost the same at about 50

live threads with the local and cloud databases. The average of 35 Daemon threads followed a similar pattern in Q1, Q8, Q12 and Q13. Overall, the performance of the local and cloud database was similar, with a very little marginal gap.

- Number of Classes Loaded: Figure D.8 shows the average number of classes loaded to execute different complicated SQL queries on local and cloud databases. The graph also compares the performance, using the response times and number of classes loaded to complete one SQL operation. The local database performance was similar to the cloud results in Q1, Q2, Q3, and Q17, while the performance of the cloud database was negatively impacted in Q6, Q8, Q9, Q11 and Q12. The number of classes loaded was highest in Q11, from 2,100 and 8,500 bytes. Overall, the performance of local and cloud database was same except for Q9 and Q11 which showed a peak load on cloud database.

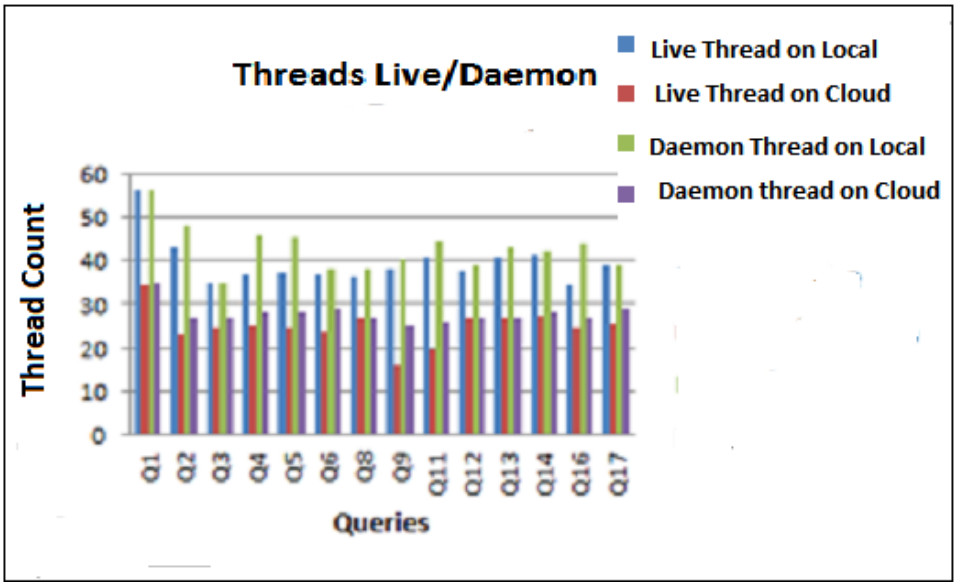

Figure D.8: Number of classes in DataNucleus JPA local versus cloud database