Google Summer of Code 2013

Apache Gora support for Oracle NoSQL datastore

Apostolos Giannakidis
ap.giannakidis@gmail.com

Mentor: Lewis John Mcgibbney
16 September, 2013

Google Summer of Code 2013
Project title: "**Apache Gora support for Oracle NoSQL datastore**"
Student name: Apostolos Giannakidis
School of Computer Science, University of Birmingham
Project Proposal: https://google-melange.appspot.com/gsoc/project/google/gsoc2013/apgiannakidis/15001
Project URL: https://issues.apache.org/jira/browse/GORA-217
Project code base: https://github.com/maestros/gora-oraclenosql

DECLARATION OF AUTHORSHIP

The report that follows has been written by, and is entirely the work of Apostolos Giannakidis.

Abstract

The recent change in enterprise data needs, the subsequent advent of NoSQL databases and their ongoing evolution have created applications that make concurrent use of various data storage engines to solve different problems. These data storage engines have different data models and different APIs. This concurrent use of a variety of databases, polyglot persistence as it is called, could lead to confusion and may become hard to manage. Additionally, some aggregate-oriented NoSQL databases are able to handle object persistence natively and efficiently, but others do not have such build-in and efficient functionality. For these reasons, a framework is needed that can consolidate the different NoSQL APIs and data models into one, and can provide a single way for object persistence in NoSQL datastores regardless of their implementation. For the relational databases this issue has been long resolved by the various ORM solutions that are available, have been proved trustworthy and have become the standard. However, in the case of NoSQL databases, there is still no standard framework. Apache Gora tries to fill this gap. Apache Gora already supports persistence to 5 NoSQL data stores.

In this project, we designed, implemented and tested a new module for integrating Apache Gora with the Oracle NoSQL Database as an object persistence data store. This module proves that an object to a key/value store mapping is possible and thus object persistence is feasible in such data stores.

Keywords: Apache Gora, NoSQL, Oracle NoSQL, ORM, Integration, Distributed Databases, Systems and Software Development

‘Οι δημιουργοί της επιστημονικής γνώσης - ερευνητές σε εργαστήρια, καθηγητές σε Πανεπιστήμια και νέοι στα πρώτα στάδια των σπουδών τους - έχουν, απέναντι στην επιστήμη, τη σχέση που έχει ο γλύπτης με τα γλυπτά του ή ένας ποιητής με την ποίηση.’

Γιώργος Γραμματικάκης (Η Κόμη της Βερενίκης)

“The creators of scientific knowledge - from the researchers in their labs to the university professors and the young people just starting their studies - stand, towards science, in the same relation as sculptors to their statues or poets to their poems.”

George Grammatikakis (Coma Berenices)

Contents

List of Figures

Listings

Nomenclature

ACID	Atomicity, Consistency, Isolation, Durability
AGPL	Affero General Public License
API	Application Programming Interface
BDB JE	Berkeley DB Java Edition
CRUD	Create, Read, Update and Delete
JPA	Java Persistence API
JSON	JavaScript Object Notation
ORM	Object-Relational Mapping
PMC	Project Management Committee
RDBMS	Relational database management system
SiTra	Simple Transformer
SLF4J	Simple Logging Facade for Java
SQL	Structured Query Language

Chapter 1

Introduction

Big Data is a revolution in the way businesses use their resources. During the last decade, businesses started collecting vast amounts of data. This data could come from everywhere: real-time data from all kinds of sensors, GPS location data, click information from web pages, or user posts from all kinds of social media website are only few examples.

Several technologies have emerged that aim to contribute to the landscape of Big Data, each one of them leveraging different aspects of Big Data. The two most important technologies that drive the Big Data revolution are MapReduce and NoSQL databases.

MapReduce, first created by Google, and further developed by Yahoo and then by the Open Source community, is a programming model that allows the processing of massive amounts of unstructured data in parallel across a distributed network of computers. Apache Hadoop is the Open Source implementation of the MapReduce programming model. Apache Hadoop is at the heart of every Big Data system and because of its increasing importance, an ecosystem of interworking tools, frameworks and applications has grown around it. Some of these are: Apache Gora, Apache Pig, Apache Hive, Apache Sqoop and Apache Avro.

NoSQL databases are not a new invention today. The term NoSQL database was first coined in 1998 [1]. We should highlight the fact that the term NoSQL does not describe a specific database system, nor a database query language. NoSQL is an umbrella term that describes all the varying data storage technologies that do not conform to the relational paradigm [1]. Non-relational technologies were created because of the realisation that the architectural concept "one size fits all", applied to datastores, did not suffice to cover the current needs of the enterprises [2]. These needs include processing of huge amounts of generated data (High Throughput), very fast access, and flexible Horizontal Scalability on commodity hardware. Also, because RDBMSs are mainly built based on the "one size fits all" concept, they include complexity that is not needed in several of the modern applications. This added complexity affects the overall performance of the database in cases where performance is the main objective. For example, in a social web site, performance and availability is more important than consistency. We are thus seeing many aspects related to the NoSQL technologies and numerous implementations that are very different than each other. This report is focused in the NoSQL field. Though it is not intended to be a thorough literature review of the various NoSQL databases and the relevant technologies, we will, in order to fully understand certain aspects of the implemented project, describe the fundamentals of the various NoSQL technologies and some characteristics of their architecture.

Today, because of the recent changes in their needs and of the advent of NoSQL databases, enterprises use a plethora of database engines in order to cover their numerous different needs. As an example, let us consider the typical scenario of a big retailer online store. Such a store could use an RDBMS to store and handle its financial data and for its warehouse reports, a key-value database to store the shopping cart data and the user sessions, a graph database for recommendations, and finally a column-oriented database for analytics and to store the user activity logs. This variety of different data storage technologies for different enterprise needs could soon become confusing and hard to manage. The term Polyglot Persistence [1] expresses just this variety of different data storage technologies that are used concurrently by the current enterprise software systems. All these different data storage technologies have different data models and different APIs.

Evidently, there is a need for a framework that will provide a single API for all the varying NoSQL data stores, along with a single data model and Object-to-Datastore mapping capabilities. This is where the Apache Gora framework comes in to fill the need. The version 0.3 of Gora supports persistence to 5 NoSQL data stores, and its community works on expanding its data store capabilities in order to achieve the project's goal, which is to become the standard data representation and persistence framework for Big Data. This project builds on top of Apache Gora and contributes towards this goal by implementing a new NoSQL data store for object persistence.

1.1 Project Scope

1.1.1 Scope statement

In this project, we will expand Apache Gora's data store capabilities by implementing a data store module that will integrate with the Oracle NoSQL Database. More specifically, Apache Gora should be enabled to persist data beans in Oracle NoSQL according to a custom mapping. The design and the implemented features of this data store will be described in detail in this report. The boundaries of this module are shown in the included Use Case diagrams.

The broad contribution to the Apache Gora Open Source community is also part of this project's scope. For this reason, and as a way to learn the implementation details of Apache Gora, several issues, bugs and improvements were identified and resolved. The details of these issues are described in chapter 5, and they are documented online in the Jira bug tracking tool that Apache maintains[1].

1.1.2 Project deliverables

The main deliverable of this project is the new Gora-OracleNoSQL module that will allow Apache Gora to use Oracle NoSQL as a persistence data store and properly utilise its features to achieve the functionality that it is required by the Gora API. The source code of this datastore module was maintained in GitHub and is publicly available here: https://github.com/maestros/gora-oraclenosql/blob/master/gora-oracle/.

Additionally, this report itself is a major deliverable that includes important information about this project. The specific sections of this report are discussed in subsection 1.4. It must be highlighted that a critical part of this project is the delivery of a detailed design and analysis of this module, not only to document it for this academic report, but also because of its Open Source nature, which means it might need to be maintained by other developers in the future.

The patches of the Apache Gora Jira issues that have been resolved are also deliverables of this project, as the contribution to the Apache Gora community was an essential part of the work.

The code for all the above deliverables, along with the report itself, are delivered on the CD that comes with this report; they are also available online, as they have been committed to the Apache Gora trunk repository.

1.2 Contribution of the project

The deliverables of this project aim to contribute to the Open Source community and specifically to the Apache Gora project. This contribution will benefit the Apache Gora project in many respects. Some of these benefits are listed below:

- Provide a new NoSQL datastore for the 0.4 release of Apache Gora.
- Make Apache Gora's datastore support larger.
- Use a popular and reliable NoSQL solution in order to gain more attention from the community and help render Gora the standard persistence framework for NoSQL databases.

[1]https://issues.apache.org/jira/browse/GORA

- Explore a different approach of a key/value database, which will extend Gora's datastore support, as the only key/value store that Gora currently supports is DynamoDB, which is a cloud database service and whose connectivity with it is performed by Web Services and not by native method calls.
- Integrate with Oracle, winner of the Big Data Company of the Year 2013 award [3] and the clear market leader in the commercial database field [4] in order to gain more enterprise users.
- Integrate seamlessly with the Oracle stack and the Big Data Appliance.
- Indirectly integrate (using External Tables) with Oracle's, industry leading, relational database.
- Take advantage of Oracle's NoSQL unique features such as: ACID transactions, simple data model, Avro JSON definitions and efficient support for Large Objects.
- Improve the overall framework by resolving major known bugs, by identifying unknown issues and by proposing and implementing new features.

1.3 Motivation

Having extensive professional experience with relational databases, and being myself a Certified Oracle SQL Expert, I wanted to explore alternative solutions to those that conform to the relational paradigm. NoSQL technologies and the various Big Data solutions are the predominant alternative and a relatively new field. Therefore, I considered a project related to the NoSQL technologies as the best choice. During my search for a suitable project topic, I came across the Apache Gora framework, which seemed a perfect fit for my knowledge needs. Additionally, I have always wanted to work on an Open Source project, to add my contribution to the work of a large community of developers, and this seemed a very good opportunity.

Apache Gora tries to solve the Polyglot Persistence issue [1] by consolidating all the different NoSQL data models and APIs into a single data model and API. Further exploring Apache Gora, I saw that it is a very new framework with great potential for becoming the standard data representation and persistence framework for Big Data because of its considerable adaptability and its modular design. However, to actually become the standard NoSQL persistence framework, it needs support for all major NoSQL datastores. Realising that at the moment, it supports only a few of them, I identified the potential for extending its integration capabilities to another datastore. While analysing the high-level requirements for such a project in order to assess its feasibility, I was very happy to see that a similar endeavour was taking place in the global event Google Summer of Code 2012. This proved that it is feasible to accomplish a similar goal within the timeframe of a summer project.

When it came to the question of which NoSQL datastore I would plan to work on, the answer came quite naturally. Since I am familiar with the Oracle ecosystem, and since Oracle NoSQL is a NoSQL database with several integration points to other enterprise systems, I considered Oracle NoSQL to be a very good candidate for Gora's new datastore. Oracle NoSQL is a pure Java-based implementation and its Community Edition is also Open Source (GNU AGPLv3). Additionally, Oracle NoSQL, being an enterprise-class database, would give Apache Gora the opportunity to be used in enterprise environments. Another point to be highlighted is that, even though Oracle NoSQL is a key-value data store, its data model is quite different from any other existing key-value database system. A final point to be made is that Oracle NoSQL has a tight integration with Apache Hadoop and of course with the Oracle Database, as well as with Oracle Coherence and Oracle Big Data Appliance. These facts make Oracle NoSQL a very interesting choice for a future Apache Gora data store, one that would be a valuable asset for the Gora framework.

1.4 Report structure

This report is composed of 9 chapters. These are organised as follows:

- **2. Background**: This chapter provides the necessary background information to properly understand the implementation characteristics of the implemented system. Additionally, it reviews the fundamental concepts and functionality of the two systems that will be integrated: Apache Gora and Oracle NoSQL.

- **3. Related Work**: This chapter presents a brief literature review and a review of similar work on the topic of Apache Gora datastore support, as well as on the topic of persisting objects in NoSQL data stores.
- **4. Requirement Analysis:** This chapter presents the results of the analysis that was performed in order to fully define the stakeholders and the system requirements of the Gora-OracleNoSQL datastore module. Finally, it presents the assessment of a feasibility study that was performed in order to evaluate the feasibility of this project.
- **5. Gora-OracleNoSQL datastore**: This chapter gives all the details of the developed module, including its analysis, design, implementation and testing. Moreover, it presents the third-party tools that were used for the creation of the developed module and also the assisting software utility that was needed for validating the results of the system. Finally, it presents a guide for creating a Gora datastore module.
- **6. Apache Gora contribution**: This chapter provides detailed information about the general contribution to the Apache Gora project that aimed its overall improvement.
- **7. Evaluation**: This chapter presents an overall evaluation of this project as well as several important aspects of its success and the methodology that was used.
- **8. Project Management**: This chapter describes the main phases and activities of this project and how they were organised. It also presents the project plan, including the milestones. Finally, it presents a risk assessment report that was created at the inception phase of this project.
- **9. Conclusion:** This chapter provides a synopsis of this project and summarises its achievements, limitations and proposed future work.

Chapter 2

Background

This chapter will present the necessary background information to make the reading of the report easier, and clarify the terms and concepts relevant to the study.

2.1 NoSQL Overview

In this section we will present an overview of the NoSQL field and introduce to the reader the fundamental concepts that are needed to understand the need of this project and its features.

2.1.1 Big Data

Before defining what Big Data is, let us first define what Big Data is not. Big Data is not a technology; not even a product. It is an umbrella term that refers to very large datasets that are challenging to handle, store, process, and analyse. Big Data is quantified Petabytes, Exabytes, and soon Zettabytes [14]. These units express the magnitude of the datasets that today's enterprises generate and handle. To be specific, the rate of data generation increases annually, monthly or even daily. Even in medium sized enterprises, the volume of data available to analyse is growing at a very fast pace.

Viewed more closely, the term "Big Data" relates to the collection of large-scale datasets so as to describe the fact that the complexity to be handled with common RDBMSs. By "handling" such datasets, we mean performing particular functions and operations over the dataset. These operations are not very different from those of the relational databases, as they include searching, transferring, sharing, analysing and of course storing information [13]. But despite the fact these functions are seemingly the same with the common data stores, there are particular requirements that have to be met if they are to efficient and effective over large-scale datasets.

Obviously, in the last decade, a great change has arisen in the **volume** of the data that is being generated and handled. However, it is not only the volume of the data that has changed, but also the type of storage used.Today, a plethora of different data systems are in production, such as the Web 2.0 applications, B2B, and Enterprise Application Integration (EAI) systems, to combine, store and handle data from different sources. The data these sources generate may include text messages, sensor data, audio, video, click streams, and log files, for just a few examples of the **variety** of the data that should be stored and handled. A third characteristic of this kind of data that has changed is the speed of their processing. There are many cases where 1 or 2 minutes of processing is considered too much. The **velocity**, as it is called, of such data demands near real-time or even real-time processing. Examples of such time-sensitive processes are fraud detection and trading systems. Doug Laney [15] in 2001 was the first to define the term "Big Data" using these three data characteristics (also called dimensions): Volume, Variety, Velocity. Now known as the 3Vs of "Big Data", they are used extensively to describe its essence.

2.1.2 Infrastructure Requirements

It is very important that we now consider Big Data from a different point of view, that of hardware and infrastructure, in order to define what the role of NoSQL databases is in the era of Big Data. As Big Data is a fairly new world in the area of applied Information Technology, this is not a regular case of common computing and infrastructure. There are particular requirements that are very unique for the case of Big Data, as they are for every computing platform.

The term Big Data is closely related to three major processes that usually occur with a strict sequence: data acquisition, data organisation and data analysis. Obviously, there would be no interest or research into Big Data if there were not the final process of data analysis. Data analysis is crucial because modern enterprises want to discover new findings or root causes by performing knowledge mining operations over Big Data. Thus, data analysis is able to resupply the chain of the three processes, by serving as the stepping stone for acquiring more data and organising them again.

Clearly, the process of storing and organising this information should be totally different than in regular data warehouses. For achieving better performance levels, there is a clear tendency to take advantage of commodity computing. This means that a large number of already available computing and hardware components are used for parallel computing in computer clusters or clouds, to get the greatest amount of useful computation at low cost [16].

Commodity computing is also related to scalability issues. There are two major types of scaling: horizontal and vertical scaling. The former is related to additions of extra machines (in terms of nodes for a distributed system), while the latter is relevant to extra resources (e.g. CPU or RAM upgrades) for a particular machine (node) of the cluster.

We can now realise that, as scalability is a very important issue for Big Data procedures, this environment is an excellent fit for NoSQL applications. This is because NoSQL databases have the right kind of design to support dynamic data structures and scalability, as they do not parse data into fixed schemas [17] and do not maintain constraints (see section 2.1.4.2). NoSQL databases are thus a very important help of in the process of data acquisition. In combination with Hadoop, NoSQL is a fully integrated solution for the other two process of data organisation and analysis.

2.1.3 CAP Theorem

At this point we have to introduce the CAP theorem, as it applies in the field of distributed systems. The CAP theorem states that when the three properties of Consistency, Availability and Partition tolerance have to be met, any distributed system can achieve only up to two of them at the same time. Of course, such requirements should be defined very clearly before proceeding with any further discussion.

By the term Consistency, we mean the validity, accuracy, and integrity of the stored data, both from logical and physical points of view. By Availability, we mean that if we can communicate with one particular node of a cluster, this node is able to read and write data. This definition is slightly different than the usual content of Availability [1]. Finally, by the term Partition tolerance, we mean that the cluster is able to react successfully to communication breakages that occur inside the cluster, that separate the cluster into different partitions unable to communication each other.

2.1.4 Data Models

Section 2.1.2 explained the importance of data organisation in Big Data. A representation of a way of organising and manipulating data is called a *data model*. A popular model, one that dominated the area of databases for many years, is the well-known relational data model.

Generally speaking, in NoSQL databases, there are no concepts of entity relationships, foreign-keys, joins, and normalisation; concepts that are the essence of the relational model that RDBMS systems use. This is because, as we described earlier, such a data model does not scale well. To solve the scalability problem while being able to access all the required information, the solution is to perform denormalisation. Denormalisation is the systematic process of introducing duplicate data into the database to improve the database's efficiency [2][22].

However, this raises two questions [2]:

1. How much denormalisation is needed?
2. How will this model work when an update happens on a denormalised field?

To answer the first question, one must start at the beginning, or even before. In NoSQL databases, an important step is performed before modelling: the definition of the queries. At this point, data modellers should ask the question "What information should be fetched?" [18]. The answer to this question will give them clues to the denormalisation granularity.

Regarding the second question, the answer relies on the application logic. Since the data are now denormalised, the database cannot guarantee their consistency. Applications are now responsible for maintaining consistent data and resolving any possible inconsistencies based on business rules.

Next, we will describe two basic characteristics of data models, as they are used in NoSQL databases.

2.1.4.1 Aggregate

The aggregate data model characteristic is adopted by various types of NoSQL databases such as key/value, document and column stores. An aggregate can be understood as a collection of data that we wish to handle as a unit. Using aggregates, NoSQL databases are able to handle operations in very large data sets in a much more efficient and scalable way than relational databases. Furthermore, aggregates are usually popular with software engineers, as they map directly to the data structures they use in their applications, and are therefore able to control data persistence in a more natural way. We should mention here that each NoSQL data model treats aggregates differently.

2.1.4.2 Schemaless

This is a major property of every NoSQL database. To understand the concept of a schemaless data model, it is convenient to compare it again with the relational data model. In order to define a fully functional relational database, one has to define the schema before it stores any data. This means that the necessary tables and their columns, relationships and constraints have to be defined in advance. Then, any data that is to be stored in the database must conform to the predefined schema.

In contrast, NoSQL databases bypass this requirement. One can simply store almost any kind of data and no restrictions are applied. However, we must stress that some kind of schema is always needed. This is because the data, in order to make sense to the application that will use them, must have a specific structure that is understandable by the application. This application-level schema is also called **implicit schema** [1]. A database that uses "Implicit Schemas" is commonly called Schemaless.

2.1.5 Types of NoSQL databases

In this section, we will briefly present three of the most important types of NoSQL databases: key/value stores, document databases, and column-family data stores.

Key/Value stores
This kind of stores is build on hash tables and is used when the access to the data is designed to be achieved by a (primary) key. This data model is composed of two columns: one for the ID (key) and one for the content (value). As this type of NoSQL database is of paramount importance to this project, we will discuss their architecture in detail in next section.

Document databases
Document databases operate with documents, which are usually expressed in XML, JSON or BSON etc. Document databases process the documents in order to retrieve and handle data that

are located inside them. It is common for such a database to store documents that have the same or a similar schema. However, this is not necessary, as each document is self-describing[1].

Column-family data stores
This type of NoSQL databases allows the user to store data with a data model similar to the key/value stores. However, each key is mapped to a value that is a set of column-families. Each column-family consists of several columns and each column is a tuple of name, value and timestamp [1]. Well-known examples of Column-Family stores are the HBase and Hypertable stores.

2.1.6 Key/Value stores

As we mentioned earlier, this type of NoSQL databases plays a major role this project. In this section, we will discuss this model in more detail and provide the necessary background on key/value stores. Some popular databases based on the key/value model are Riak, Amazon's DynamoDB, Oracle NoSQL, and Project Voldemort which is an open-source implementation of Amazon DynamoDB.

2.1.6.1 Data model

This model is the simplest and probably the most popular of all the NoSQL data stores. As we said, each record is represented by a key and a value column. The value is in essence the aggregate and the key is the index to the aggregate. The user is allowed to retrieve the value for the key, insert values for particular keys or delete a key from the store.

2.1.6.2 Key structures

Despite the fact the general concept of key/value store is the same for most of the databases, there are different implementations of the key structure in different NoSQL databases. For example, the key structure in the Oracle NoSQL Database is based on the major and minor key paradigm, while DynamoDB uses either one hash attribute or one hash and range attribute for the key.

2.1.6.3 Value structures

Values usually represent an opaque data type. That means that values are black box records for the key/value store. The model is not interested in the nature of this data as values. Query operations based on value are only allowed at the application level. That means that the application retrieves the records, iterates through them and discards those who do not fulfil the value criteria. Thus, key/value store databases are not able to perform queries based on values, as these values are opaque to the database.

Additionally, as we mentioned earlier, the values do not have any schema restrictions. Each record could have values of any content. Finally, it is important to mention that usually there is no limit on the value part. The only limit is the one imposed by the Operating System.

2.1.6.4 Avro data format

Apache Avro is a data serialisation system which provides rich data structures and an efficient binary data format. Avro names these data structures *schemas*, and are defined with JSON [20]. The Avro specification allows a plethora of data types for each schema, which includes various primitive types such as boolean, int, float, bytes and complex types such as records, enums, arrays and maps. Using Avro, one is able to compile a schema into generated code.

2.1.6.5 Range queries

As we have described above, there are plenty of implementations of key/value stores. While the goal of all these solutions is to offer highly available and scalable data storage services, they have significant differences amongst each other in terms of the architecture, their features, their use cases and even their internal data models [12].

Range queries provide a crucial functionality to a key/value store: to be able to identify one or more keys that are stored, based on specific criteria. Such criteria are usually a start key (lower limit) and an end key (upper limit). These two criteria define a specific range of keys that can be used to access one or more key/value pairs in the database.

Range queries are an important feature for every data store that is used by OLTP applications. It may be a strange observation to someone unfamiliar with the NoSQL field, but the feature of range queries is not available to all key/value store implementations [12]. An extension of this concept, of not only single key/value pairs but at the level of whole persistent objects, is provided by the Apache Gora framework (see section 5.4.5).

2.2 Oracle NoSQL

2.2.1 Overview

The Oracle NoSQL Database is a distributed, highly scalable, highly available key-value data store. It is focused to address the low latency data access needs that are required in processing "Big Data" volumes [7]. "Key-value data store" means that the database stores and retrieves the data which is organised into key-value pairs.

It is based on the proven and highly-tested Berkeley DB Java Edition and is a pure Java-based implementation. On top of the BDB JE, it provides a simple data model based on multi-component major and minor keys and full Create, Read, Update and Delete (CRUD) operations, with adjustable durability and consistency transactional guarantees. Additionally, it has tight integration with the Apache Hadoop MapReduce infrastructure, with the relational Oracle database (RDBMS), with Oracle Coherence and with the Oracle Big Data Appliance.

It has several use cases, including social networks, customer profile management, real-time sensor aggregation, content management, session management, archiving and many more. The Oracle NoSQL Database can be used by any application that requires network-accessible key-value data, as the back-end database in a three-tier architecture [8].

It is released in two editions: the Enterprise Edition and the Community Edition, which is Open Source and licensed under the Affero General Public License version 3. It is part of the Oracle Big Data Appliance and even though it was released 1.5 year ago, it already has several thousands of deployments in enterprise environments.

In the following sections, we will discuss the architecture, the data model and the API of the Oracle NoSQL Database. Please note that these sections will not cover all the aspects, properties and features of the Oracle NoSQL database, but only those that were used by the Gora-OracleNoSQL module and/or those that are needed to understand the rationale behind some of the key implementation decisions of that module.

2.2.2 Architecture

The Oracle NoSQL Database, being a distributed database, is organised *in Storage Nodes* and *Replication Nodes*. A Storage Node is a physical (or virtual) machine that hosts one or more Replication Nodes. Conceptually, a Replication Node can be perceived as a single database which contains specific key-value pairs. In the background, the Oracle NoSQL Database Driver partitions the data and evenly distributes it across nodes, on the basis of on a consistent hashing algorithm applied on the key. One or more Replication Nodes constitute a *shard*. One of the Replications Nodes of a shard is elected to be the *Master Node*, which is responsible for handling the write operations and replicating the write operations to the other Replication Nodes of the shard, which are called *replicas*. The replicas handle the read-only operations. In essence, this means that by having more shards (and thus more master nodes) there is higher write throughput, and by having more replicas in a shard there is higher read throughput. The term *Replication Factor* refers to the number of nodes that a shard has.

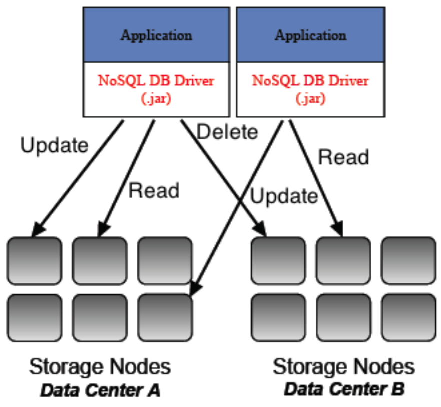


Figure 2.1: Oracle NoSQL Database architecture

The data model of Oracle NoSQL will be explained in more detail in the section 2.2.3. However, we must understand at this point that the key part of a key-value pair is composed of the Major key and the Minor key. It is important to highlight the fact that Oracle NoSQL provides ACID guarantees across multiple data pairs depending on their major and minor keys. The single-master, multi-replica architectural model of Oracle NoSQL allows it to provide highly available database replication with flexible durability policies per transaction. This flexibility is exposed at the API level. This allows the application developer to make the proper decision regarding the tradeoff of performance and consistency/durability for each CRUD operation or transaction. It should be noted that other single-master, multi-replica distributed systems do not provide this flexibility. Instead, they provide system-wide configurations, which are very limiting for the application developer who needs control over the data operations.

An important characteristic that needs further elaboration at this point is that Oracle NoSQL achieves very fast indexed key lookup because it provides data locality for all the keys that share the same Major key. Therefore, if application developers are to exploit properly the fact that key-value pairs are evenly distributed across the nodes, they should properly design their application's data model and make wise use of Major key components. Having wisely-chosen Major keys has another important benefit: multiple CRUD operations on multiple key-value pairs that share the same Major key are executed under a single atomic operation with ACID guarantees.

The store records versions for each modification of the key-value pairs. However, it should be remembered that it maintains only a single version of them: the latest. Therefore, there is no need for conflict resolution. In essence, this means that the application developers do not have to reconcile incompatible versions. This is achieved because of the single-master, multi-replica architecture. The master node, always only one for each shard, always stores the latest (most up-to-date) value for a specific key. Of course, the replicas, which are always read-only, might have a slightly older version, depending on the configured consistency level. The reason that Oracle NoSQL records versions for each key-value pair is to be able to provide read-modify-write operations while consistency is preserved.

2.2.3 Data model

In this section, we will describe the data model that Oracle NoSQL Database uses. It is very important to understand this data model in order to be able to understand the data model that we created for the Gora-OracleNoSQL datastore module, which exploits the characteristics of the one used by Oracle NoSQL.

The Oracle NoSQL Database stores its data using key-value pairs. In the Oracle NoSQL terminology, a key-value pair could also be called "a record". A key-value pair is a map data structure in which the *key* references the associated *value*. A key follows the Major+Minor-key paradigm. This means that a key is composed of a Major part and a Minor part. Each part (major and minor) has one or more components, specified as an ordered list. All the components that compose the Major and the Minor parts are of type String. The concatenation of all the Major and Minor key components is called the *full key*. Note that the Major key is mandatory and must have at least one component. The Minor key is optional. Multiple key-value pairs that share the same Major key but different Minor keys create sub-structures with an organisation similar to a directory path specification in a Unix file system. An example of this structure could be visualised as: /Major1/Major2/Minor1/Minor2/Minor3/.

The value part of the key-value pair is an opaque data value. This means that the value is byte arrays of arbitrary length that are uninterpreted by the database. The database does not know the structure/schema of the value, nor it does have query access based on the value part of its stored key-value pairs. The serialisation of the data structures that need to be stored as byte arrays and their deserialisation is left to the application. However, Oracle NoSQL has Avro bindings and recommends to serialise the values using Avro instead of a custom serialisation format. As a final note regarding the values, it should be mentioned that there are no restrictions on the size of the values. However, consideration should be given to how large the key-values should be, because this directly affects the datastore's performance.

The following figure [1] illustrates an example of two key-value pairs that share the same Major key but have different Minor keys, for a clear understanding of the data model used by Oracle NoSQL.

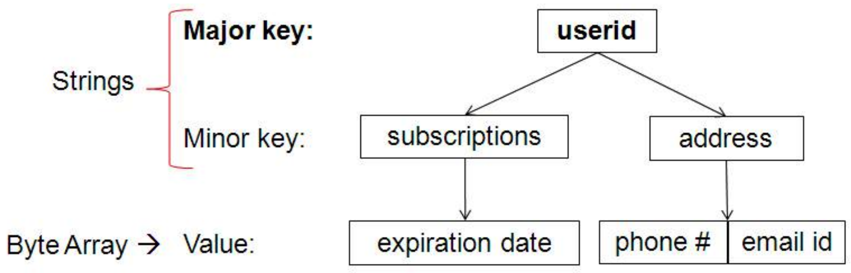


Figure 2.2: Oracle NoSQL data model example

2.2.4 API

The Oracle NoSQL API for manipulating key-value pairs is straightforward. All the available classes and methods of the API are made available to the application developer in the kvclient.jar file. The following will discuss very briefly the main methods of the API. The methods presented are grouped based on their operation type.

Basic CRUD operations

There are 3 main methods that provide CRUD operations, but there are several variations of each of them. The variations of these methods are employed for specific use cases. Therefore, we will only present here only those that were actually used by the Gora-OracleNoSQL datastore module.

- get(Key key) → Value
  Retrieves the value corresponding to the given key. A call of this method retrieves the value of only one key-value pair at a time.
- put(Key key, Value value)
  Inserts the persistent object with the given key if the key does not exist. If an key-value pair with the given key already exists, the value of that key-value pair will silently be replaced.
- delete(Key key)
  Deletes from the database the key-value pair that corresponds to the given key.

[1] Figure borrowed by the Oracle NoSQL documentation. Available at: http://www.oracle.com/technetwork/products/nosqldb/overview/key-value-497224.html

- multiDelete(Key parentKey, KeyRange subRange, Depth depth) → int
  Deletes the descendant Key/Value pairs associated with the parentKey and returns the number of the keys that were deleted.

Iteration
Apart from the above-mentioned basic CRUD operations, Oracle NoSQL Database provides additional API methods for two types of iteration: unordered and ordered iteration over records. Ordered iteration is performed only for records that have the same full Major key. Unordered iteration is performed when it is given only partial Major key or no key at all. The following are the related methods that were used:

- multiGet(Key parentKey, KeyRange subRange, Depth depth) → SortedMap<Key,ValueVersion>
  Returns the descendant key/value pairs associated with the parentKey. A call of this method retrieves multiple, sorted, key-value pairs at a time, as long as they all share the same Major key.
- storeKeysIterator(Direction direction, int batchSize) → Iterator<Key>
  Returns an Iterator which iterates over *all* keys in unsorted order. Since this method operates in multiple key-value pairs that do not share the same Major key, the result does not have transaction semantics.
- multiGetKeysIterator(Direction direction, int batchSize, Key parentKey, KeyRange subRange, Depth depth) → Iterator<Key>
  Returns an Iterator over a SortedSet of descendant keys associated with the parentKey.

Bulk Operations
Another type of operations, apart from the basic single-record CRUD operations, are the operations that are bundled together into a collection of operations in order to be executed transactionally all together, as long as they all share the same Major key.

- execute(List<Operation> operations) → List<OperationResult>
  Executes the operations in the given list of operations in an efficient way within the scope of a single transaction. The sequence of operations should be associated with keys that share the same Major Path.

2.3 Apache Gora

2.3.1 Overview

In the relational database world, there are plenty of Object-Relational-Mapping frameworks that solve the object-relational impedance mismatch problem. Hibernate [2], that implements the JPA 2.0 specification, has become the standard in this field [3]. ORM frameworks, such as Hibernate, create an abstraction layer for data storage. Therefore, using such a framework, it is in fact feasible to change the back-end database system with minimal (or even no) changes in the application code.

However, in the NoSQL world, there is no similar standard specification for data storage abstraction and object persistence. ORM frameworks, such as Hibernate, are not sufficient to handle the full power of NoSQL databases. This is where Apache Gora comes in to fill this gap. Apache Gora is primarily, but not only, a NoSQL storage abstraction framework. Apache Gora is one of the very few object persistence frameworks for NoSQL datastores, and according to our research, it is the only one specifically built for this [5]. More specifically, the Apache Gora framework provides an easy-to-use in-memory data model and persistence for Big Data and supports functionality for persisting to column stores, key value stores, document stores and RDBMSs, and analysing the data with extensive Apache Hadoop MapReduce support. Gora's overall goal is to become the standard data representation and persistence framework for Big Data [6].

The Apache Gora first entered the Apache Incubator on September 2010 and became a Top Level Project in January 2012. Initially, the project started as an object persistence framework

[2]http://www.hibernate.org/
[3]http://jcp.org/aboutJava/communityprocess/final/jsr317/

for column oriented databases such as Apache HBase [4] and Apache Cassandra [5]. However, it was soon evident that this framework could be extended to support any other kind of NoSQL database. In its current release it supports 5 NoSQL databases and many others are implemented by the members of its community. More specifically, it supports Apache HBase, Apache Cassandra, Apache Accumulo [6], Amazon DynamoDB [7]and a store for Apache Avro data files [8].

Apart from its object persistence capabilities, Apache Gora takes advantage of parallel computation by having extensive out-of-the-box support for Apache Hadoop MapReduce. Every Gora data store that implements the MapReduce-specific API methods, can be used as input and output of Hadoop jobs [9].

As a final note, it should be mentioned that Apache Gora already has several thousands of deployments, mostly as a dependency of Apache Nutch, a scalable web crawler.

2.3.2 Architecture

In this section we will present a high level overview of the Gora Architecture. Gora is a multi-module project. Each supported datastore has its own module, and the core functionality of Gora has its own separate module.

Gora-Core is the module where the core Gora interfaces and functionality are located, as well as the AvroStore, the MemoryStore, and the testing suite. The core functionality includes the persistency data structures, the query functionality, the Gora Compiler, the MapReduce supported functionality, and some utility classes that are used by gora-core and the datastore modules.

All the other modules are datastore modules, each one specific to one NoSQL database. As discussed in the previous section, Gora is able to use several different back-ends transparently. Each datastore implementation is independent from the others as long as they implement the required methods of the DataStoreBase core abstract class. Each datastore module comes with a gora.properties file that should be changed by the application developer who wants to use the specific Gora datastore. This file contains required configuration for the specific datastore.

Each data structure that we want to persist in Gora, instead of being written as a Java class, should be defined in Apache *Avro* format. Gora uses Avro to define the data structures to be persisted and also for serialisation of these Avro structures. According to the Avro terminology such a definition is called an Avro Schema and is declared using JSON.

Apache Gora has a compiler, the *GoraCompiler* (which extends the Avro Specific Compiler) to generate the necessary Java data beans from the Avro Schema. The data bean not only contains the fields as specified in the Avro Schema, it also contains the necessary fields to track the persistency state of the data objects. For this reason, each generated data bean extends a PersistentBase class.

The main data structure that handles the actual object persistence is the *DataStore*. A DataStore object is responsible for retrieving, persisting, deleting and querying persistent objects. Note that each DataStore object is always associated with a key and a value class. The value class is the persistent class; i.e. the class of the objects that should be persisted. Therefore, a DataStore object is capable of handling the persistence of only one type of persistent object (data bean). The types of these classes (key class and value class) are defined during the creation of the new DataStore object. The value class is generated using the GoraCompiler, as specified above. DataStore objects are generated by the DataStoreFactory. Every call to the DataStoreFactory.getDataStore() reads the gora.properties file from the CLASSPATH to identify the Gora datastore module that should be used for persistence and other datastore-specific parameters. It then initialises and returns a new DataStore object. In order to handle more than one data bean, multiple DataStore objects should be created, one for each data bean.

[4]http://hbase.apache.org/
[5]http://cassandra.apache.org/
[6]http://accumulo.apache.org/
[7]http://aws.amazon.com/dynamodb/
[8]http://Avro.apache.org/
[9]http://hadoop.apache.org/docs/stable/api/org/apache/hadoop/mapred/jobcontrol/Job.html

After the Gora data bean is generated by the GoraCompiler, Apache Gora needs a way to know where in the datastore it should persist instances of this data bean and its fields. This is the work of the mapping, which is a file in XML that gives important information to Gora such as the class for the key, the name of the database table to persist instances of the data bean, what the primary key is, and the names of the fields and under which name they should be persisted in the database. Note that each datastore has its own, specific, structure for the mapping file. This is because each NoSQL database has its own, unique, structure and features. By having datastore-specific mapping files, Gora takes advantage of these unique datastore structures and features.

A final note is needed in this section to clarify what a *datastore schema* means in the context of Gora. Gora does not provide a formal definition of what a datastore schema is (or should be). For this reason, we had to deduct its meaning and purpose by studying the implementation of the Gora core functionality and of its various datastore modules. The reason for this confusion is because that is dependant on the back-end datastore. Thus, we came to the conclusion that a Gora datastore schema is just the container in which the data of the persistent objects are stored. The container is datastore specific and could be any structure that fits this requirement. For example, it could be a table, a column family, a key component, or any other container structure provided by each of the back-end database. Also note that a Gora datastore schema contains data of only one data bean and not data of every data bean available to Gora. This is because, as described earlier, a datastore handles only one type of persistent object (data bean).

2.3.3 API

The Gora API is pretty big, as it not only contains classes and methods for object persistence, but also for MapReduce and several datastore-specific methods. In this section we will briefly describe the main API calls provided by DataStore objects in order to better understand the Gora core functionality. Following are the methods, grouped by their operation type:

CRUD operations

- get(K key) → T
  Returns the Persistent object corresponding to the given key.
- get(K key, String[] fields) → T
  Returns the Persistent object corresponding to the given key with the specified fields.
- put(K key, T obj)
  Inserts the persistent object with the given key. If an object with the same key already exists it will silently be replaced.
- delete(K key)
  Deletes a persistent object from the database.
- flush()
  Every CRUD operation can be accumulated before they are actually executed in the back-end database. This method forces the accumulated operations to be flushed.

Query operations

- newQuery() → Query<K, T>
  Constructs and returns a new Query. A Query can search for a specific key or a range of keys. The results of a Query can be iterated with the Result interface.
- execute(Query<K, T> query) → Result<K,T>
  Executes the given query and returns the results in a Result object.
- deleteByQuery(Query<K, T> query)
  Deletes all the objects matching the query. If the query specifies specific fields, only those fields are deleted, instead of the whole persistent objects.

Schema operations

- createSchema()
  Creates the schema in the datastore to hold the objects. If the schema is already created previously, the operation is ignored.

- deleteSchema()
  Deletes the schema in the datastore that holds the persistent objects. After the execution of this method, no persistent objects exist in the database.
- schemaExists()
  Checks if the schema exists or not.

Chapter 3

Related Work

As described in the scope statement in section 1.1.1, this project has two main goals. The first is to integrate Apache Gora with Oracle NoSQL, and the second is to persist objects in Oracle NoSQL. In this chapter, we will briefly review the Gora-DynamoDB module that is another datastore module that already integrates with the Apache Gora framework. We will also briefly present another object persistence solution for NoSQL databases.

This research of related work was very valuable at the beginning of this project, as it enabled us to assess the feasibility of this project and also learn how others managed to achieve similar goals. This list is by no means complete. It includes the most important projects that we identified that have detailed documentation.

3.1 Apache Gora integration

Apache Gora is able to persist objects in several NoSQL databases. In this section, we will briefly describe only one of them: the Gora-DynamoDB datastore module, which has many similarities with our own project. First and foremost, the Gora-DynamoDB project was also accepted in the Google Summer of Code 2012, and was completed on time. Second, DynamoDB is a key/value data store, as Oracle NoSQL is. In the following section, we will give a high-level overview of the Gora-DynamoDB datastore module.

3.1.1 Gora-DynamoDB

The Gora-DynamoDB module was originally motivated by two facts. First, Cloud computing has become very popular and its popularity continues to rise exponentially among IT departments. Second, most of big data enthusiasts do not have access to powerful Hadoop clusters, nor to sophisticated NoSQL data store setups.

Amazon's DynamoDB is a very popular Cloud key/value store and was considered a good datastore for Apache Gora. This will enable Apache Gora developers to use Amazon's popular NoSQL datastore without having to worry about data modelling and hardware.

The Gora-DynamoDB module handles serialisation as others modules inside Gora, although it relies only on a XML file, as it has no direct dependency on Apache Avro. There are some differences on the way data is flushed to the back-end data store as when this module was implemented there was an advantage in reading/writing a single object or a batch of a hundred elements. Inside regular data stores, the advantage of holding records in memory is a lower usage of networking resources, which can be the bottleneck in distributed applications.

Finally, it should be mentioned that the Gora-DynamoDB module made several changes to the Gora abstraction layer, because it now needed to support not only disk-based datastores but also datastores based on web services (such as Amazon's DynamoDB). Now that such a layer exists, that supports web-service based datastores, other modules can be created that will add support to other popular Cloud NoSQL databases such as Google App Engine, Microsoft Data Services (Azure), and others.

3.2 NoSQL Persistence Frameworks

Apache Gora is a working framework for object persistence in NoSQL databases. During our research for related works, we identified another project that is still under research but has published documentation of its architecture. In the following section we will briefly discuss this research framework.

3.2.1 Object-NoSQL Datastore Mapper

Cabibbo [21] proposed a framework for managing persistent objects in NoSQL systems. This system allows software engineers to take advantage of an ORM-like API which is similar to JPA. This framework is based on a data model that consists of entities, relationships and embeddable objects. Interestingly, even though it is not released yet, it already supports access to a large variety of NoSQL systems such as Cassandra, Couchbase, MongoDB and Oracle NoSQL, which proves the flexibility of the framework to support new NoSQL datastores.

It is designed in a way that allows the system to be independent from the specific datastore, and that facilitates the storage of different entities based on multiple representation strategies. Its architecture is based on three major layers: the API, the Internal Representation Management and the Datastore Access. The first layer (API) can be used by developers and is based on an extended version of JPA. The second layer is responsible for the internal management of the entities within a current transaction, while the third is responsible for connectivity via *connectors* to different NoSQL datastores. To describe the structure of the persistent objects it uses annotations, which are exposed at the API level. CRUD operations are provided by the *entity manager* interface that is defined in the Java Persistence API (JPA). Each Internal Representation Manager is responsible for implementing this interface. A final point of interest is that the Datastore Access layer relies on different *entity representation strategies*. Such as strategy defines how an internal representation of a persistent object can be stored in a NoSQL database. Each connector implements one such strategy. This effectively allows to have multiple connectors for the same back-end NoSQL datastore and thus different entity representation strategies for the same datastore.

Chapter 4

Requirement Analysis

In this chapter, we will perform a requirement analysis for the Gora-OracleNoSQL datastore module which is the main objective of this project. Specifically, we will identify the stakeholders, define the system requirements and classify them by type (functional / non-functional), and finally we will assess the feasibility of such a module.

4.1 Stakeholder identification

Before we proceed to define the system requirements, we need to identify the stakeholders. Understanding who is affected by the system will help us make the system requirements (defined in section 4.2) more specific, targeted and clear.

The following persons, groups of people, or organisations are affected by the implemented system:

- Decision-makers:
  - Datastore module developer (Primary decision-maker)
  - Apache Gora project community
- Users:
  - Application developers (Primary users)
  - IT Architects
- Sponsor:
  - Google Corporation
- Wider environment:
  - Apache Software Foundation
  - Oracle Corporation

We should note that the actual users of this system (Gora-OracleNoSQL datastore module) are themselves developers, who develop their applications using Apache Gora and this module as the back-end data storage mechanism to achieve object persistence.

We should also note that even though the Gora project community are classified as "Decision makers", they did not actually made any explicit decisions regarding the Gora-OracleNoSQL module. They are classified as "Decision makers" because of their indirect, implicit influence on parts of the module's architecture; specifically, because several of the module's classes had to implement specific Gora interfaces, or extend specific Gora classes, as defined by the Gora project community.

Finally, we should note that Google Corporation is the sponsor of this project and has participated in the funding because this project is part of the Google Summer of Code 2013 event.

4.2 Requirements specification

One of the fundamental steps towards the successful completion of any project is to elicit the proper requirements of the system to be implement, during the analysis phase of the project. The aim of this section is to identify the system requirements and constraints that drive its design and its implementation. The requirements are classified based on their type (functional / non-functional).

Note that with the term *system,* we are referring to the Gora-OracleNoSQL datastore module and *not* to the Gora framework itself. With the term *persistent object,* we are referring to the object that could be stored and/or retrieved to/from the datastore.

4.2.1 Functional requirements

We considered the following functional requirements (FR). Each functional requirement has been assigned an identifier in order to be easily referred to in other parts of the analysis and design sections. For each high-level functional requirement, we have identified its rationale and also performed a breakdown analysis to identify its sub-requirements.

FR. 1: The system should be properly integrated with the Apache Gora framework.
Rationale: To achieve the main project's goal: Apache Gora should be able to use this system as a fully supported datastore module.
a) The system should use the latest API of Apache Gora.
b) The system should extend Gora's core classes and implement its core interfaces.

FR. 2: The system should be properly integrated with the Oracle NoSQL database.
Rationale: To achieve the main project's goal: to use the Oracle NoSQL database as the backend datastore of this system.
a) The system should use the latest API of Oracle NoSQL database.
b) The system should make efficient use of the provided functionality and data model of Oracle NoSQL database.

FR. 3: The system should provide basic CRUD operations for persistent objects based on their keys.
Rationale: To achieve object persistence.
a) The system should properly serialise/deserialise the persistent objects based on their types.
b) The system should be able to access/persist/retrieve speficic fields of the persistent objects.

FR. 4: The system should provide extended Get and Delete operations based on the persistent objects and not only based on their keys.
Rationale: To achieve more dynamic operations and extended functionality.
a) The system should be able to identify which field of the persistent object represents its key, according to the datastore specific mapping.
b) The system should acquire the value of the key of the provided persistent object using reflection.

FR. 5: The system should provide support for range queries.
Rationale: To achieve batch CRUD operations based on range queries.
a) The queries should operate based on a single key or a range of keys.
b) The system should allow the creation of new queries.
c) The system should allow the execution of queries.
d) The system should be able to perform range queries to retrieve single or multiple specific persistent objects.
e) The system should be able to retrieve multiple persistent objects while preserving the natural order of their keys.
f) The system should be able to perform Delete single or multiple specific persistent objects based on range queries.

FR. 6: The system should provide configurable consistency & durability for the CRUD operations.
Rationale: To exploit the flexible capabilities of consistency & durability that Oracle NoSQL database provides.
a) The consistency and durability settings should be defined in the gora.properties file.
b) The system should provide sensible defaults for consistency and durability properties.

FR. 7: The system should be configured, if required, by proper use of the gora.properties file.
Rationale: To achieve a fully customised behavior of the system operations.
a) The system should be able to locate, access and parse the gora.properties file.
b) The system should be able to identify specific properties of the gora.properties file and read its values.
c) The system should be robust enough not to terminate abnormally or accept invalid or missing values. Instead it should use sensible defaults [10].

FR. 8: The system should provide a custom mapping between the Gora data model and the Oracle NoSQL data model.
Rationale: To achieve a bidirectional translation of the two incompatible data models.
a) The system should use XML to define the mapping.
b) The system should be able to locate, access and parse the XML mapping file.

FR. 9: The system should provide transactions with ACID guarantees, for multiple CRUD operations of the same persistent object.
Rationale: To achieve isolation between possible concurrent access of the same persistent object and improve the overall performance.
a) The system should accumulate the called CRUD operations in the order in which they were called.
b) The system should provide a way to execute the accumulated operations in the order in which they were accumulated.
c) The system should make use of the proper method calls of the Oracle NoSQL API to achieve ACID transactions.

FR. 10: The system should be able to connect to multiple Oracle NoSQL master nodes.
Rationale: To achieve system robustness and node failure resilience.
a) The system should be able to be configured for the addresses of the multiple master nodes by reading the gora.properties file.

FR. 11: The system should provide a Result Cache mechanism.
Rationale: To achieve better read throughput in case of frequent access of the same persistent object.
a) The system should store in memory the latest retrieved persistent object and use the memory-stored one if the same key is needed in a subsequent call, instead of making a new database Get request.

FR. 12: The system should provide an embedded version of the Oracle NoSQL database server.
Rationale: Since Oracle does not provide one, a custom embedded Oracle NoSQL server should be created to be used during the automated execution of test cases.
a) The system should be able to spawn a new operating system process and execute the Oracle NoSQL server.
b) The system should be robust in case of failure to start the Oracle NoSQL server and provide error handling and retrying functionality.
c) The system should be able to properly terminate the process and release any opened resources.

FR. 13: The system should be able to persist fields of any datatype available in the Avro format.
Rationale: Gora data beans use Avro for serialisation and any of its supported datatypes should be persisted.
a) The system should be able to persist any primitive datatypes.
b) The system should be able to persist Avro complex types including Records, Unions, Enums, Maps and Arrays.

4.2.2 Non-functional requirements

We considered the following non-functional requirements (NFR). We will present them according to the classification proposed by Dr.Ian Sommerville[1] as presented in his Software Engineering book [9].

NFR. 1: The system should use the functionality of the existing libraries provided by Gora-Core and its dependencies, where possible.
Organisational (Development) - Implementation

NFR. 2: The system should use the functionality of Open Source libraries only.
Organisational (Development) - Implementation

NFR. 3: The system and its documentation should be released under the ASL2 (Apache Software License 2.0).
External (Legislative) - License

NFR. 4: The system should adhere to the same architectural design [9] as the other Gora datastore modules and follow the same Design Patterns, where available.
Organisational (Development) - Design method

NFR. 5: Progress reports presenting details about the effort expended on each system feature must be produced every month.
Organisational (Operational) - Process

NFR. 6: The system should be documented properly and thoroughly in order to be easily maintained by the Open Source community.
Product (Maintainability) - Documentation

NFR. 7: The system should be implemented in Java, because the Gora framework offers its API only in Java.
Organisational (Environmental) - Programming language

NFR. 8: The system should be highly portable, running on a range of Linux/Unix platforms and Windows.
Organisational (Environmental) - Portability

NFR. 9: The system should provide a reasonable response time.
Product (Efficiency) - Performance

4.3 Feasibility assessment

During the analysis phase of this project we assessed the feasibility of this system, on the basis of a careful consideration of the project's scope, the defined requirements, the technical characteristics and the architecture of the two systems that had to be integrated. Two other important aspects that were taken into consideration for assessing the project's feasibility were the project's timeframe and our lack of knowledge in the NoSQL field.

After taking account of the above, we considered the system to be feasible for the following reasons:

- The project's timeframe is considered adequate for the successful completion of this project's objectives. This is because another Gora datastore module was created as part of the Google Summer of Code 2012; thus similar objectives were achieved in roughly the same duration.
- Even though Apache Gora had no support of other key/value store with native method calls, it supports 5 other different NoSQL databases, which proves that its API and its overall architecture is flexible enough to allow the integration with a key/value store such as Oracle NoSQL.

[1]http://www.software-engin.com/

- As indicated in the Related Work section, both systems have been successfully integrated with a plethora of other systems, which proves their integration capabilities.
- Apache Gora has specific mappings for each back-end datastore. This means that, regardless of the data model of the back-end database, by designing a proper mapping scheme, Gora should be able to translate the Gora data model into the data model of Oracle NoSQL. In the extreme case of not being able to achieve such a custom mapping scheme, me and my supervisor had already established an alternative solution. The alternative solution would be to use a model transformation tool such as SiTra [2], which is a tested solution, and I would have extensive support from my University's supervisor who is also responsible for the development of this tool.
- The lack of knowledge in the NoSQL field is a definite risk for the successful completion of the project. However, the fact that we allocated about 1 month in the beginning of the project in which a significant amount of studying will take place and the fact that I would receive the guidance of my supervisor, reduces (if not eliminate) the risk.

For the above-mentioned reasons, we considered the project to be feasible. However, as it was apparent from the above, there were some risks inherent in this project. For this reason, a proper risk analysis took place in order to reduce and/or properly circumvent them. The risk analysis can be found in the section 8.3.

[2]http://www.cs.bham.ac.uk/∼bxb/Sitra/

Chapter 5

Gora-OracleNoSQL datastore

5.1 Overview

The creation of the Gora-OracleNoSQL data store module is the main objective of this project. Its purpose is to allow the Gora framework to use the Oracle NoSQL database as a data store for persisting objects (Gora data beans). The Gora-OracleNoSQL data store module has two main integration points: one with Apache Gora and one with the Oracle NoSQL database. It implements all the necessary functionality as required by the Gora API contracts (interfaces) and it provides some extra functionality to achieve several purposes, as will be described in the following sections.

An analysis of this module was performed in chapter 4, which presents in significant detail the stakeholders and the functional and non-functional requirements of the module. The design, the implementation and the test cases were based on these requirements. In this chapter, we will present the main features of the Gora-OracleNoSQL module (referred as *system* from now on) and its design. Furthermore, we will discuss the implementation details of its main features and provide an overview of how it was tested. Finally, we will briefly present the assisting software that was implemented for the needs of the demonstration of this system.

5.2 Features

Following is a full list of the main features that the system provides. For each feature, we will provide a brief overview of its functionality. Also, for some of the features we will provide sample code snippets of how they can be used. It is important to understand that these code snippets are ***not*** parts of the code of the Gora-OracleNoSQL datastore module. They are examples of how other developers can make proper use of the Gora-OracleNoSQL datastore module.

- **Full CRUD operations**
  Gora-OracleNoSQL provides full CRUD operations for persisting objects. The CRUD operations are fully compatible with the Gora API. An important feature that should be highlighted here is that the module is able to **retrieve specific fields** of the persistent objects. This allows a fine-grained access to persistent objects, which is a powerful feature for the application developer. An example of a DataStore get operation to retrieve specific field could be the following code snippet:

Listing 5.1: Usage of DataStore.get(K key, String[] fields)

```
OracleStore<String, Pageview> dataStore;
Configuration conf = new Configuration();
dataStore = DataStoreFactory.getDataStore(String.class, Pageview.class, conf);
String[] fields = {"ip", "timestamp"}; dataStore.get("1", fields);
```

- **Query operations**
  Another feature that is provided is the ability to create queries. These queries can be used to search the database for a single persistent object, or for multiple persistent objects, based on a key range. The identified persistent objects can then be used to either be retrieved (Get operation) or be deleted (**DeleteByQuery(Query<K, T> query)** operation). Query operations are mostly useful in conjunction with Hadoop jobs.
- **Extra API functionality**
  Apart from the implemented CRUD functionality that is required by the Gora API, some extra operations where implemented that provide added value to the overall capabilities of the system. More specifically, the system is able to check the existence of a persistent object (**exists(T obj)**) in the back-end database given as an argument the object itself, not just the key of the persistent object. Moreover, with the same process, a persistent object can be deleted (**delete(T obj)**) from the back-end database given as an argument the object itself. An example of usage of these two methods can be seen in the following code snippet:

  Listing 5.2: Usage of DataStore.exists(T obj) and DataStore.delete(T obj)

  ```
  WebPage myWebPage;
  myWebPage= dataStore.get("www.google.com");
  if (dataStore.exists(myWebPage))
    log.info("persistent object exists.");
  dataStore.delete(myWebPage);
  if (dataStore.exists(myWebPage))
    log.info("persistent object exists.");
  else
    log.info("persistent object does not exist.");
  ```

- **ACID transactions, where possible**
  The Oracle NoSQL Database provides ACID transactions, depending on the keys of the operations, as described in section 2.2.2. The Gora-OracleNoSQL module accumulates the called CRUD operations and executes them, in the same order in which they where called, during the **flush()** operation. In this way, isolation is achieved between possible concurrent access of the same persistent object. It also improves the overall performance.
- **Configurable consistency & durability**
  Being able to control the level of consistency and durability is a very important factor for the success of a distributed system. The Oracle NoSQL Database, in contrast to many other NoSQL databases, provides variable consistency for read operations and varying degrees of durability for update operations. The Gora-OracleNoSQL module exploits this feature of Oracle NoSQL Database and itself provides the same variable levels of consistency and durability. These can be configured globally in the gora.properties file.
- **Node failure resistance**
  The Gora-OracleNoSQL module is able to connect to multiple master nodes of the Oracle NoSQL data centres. This node failure resistance feature is valuable for enterprise infrastructures that have multiple data centres. The addresses of the multiple master nodes can be configured in the gora.properties file.
- **Preservation of key natural sort order**
  The Oracle NoSQL Database supports keys only of type String. This may become a problem in case the Gora data bean must be persistent using another data type, such as Long. The Gora-OracleNoSQL module stores the key as a String, however, it is encoded in *base32hex* [1]. This encoding allows multiple String keys to be retrieved in their natural key order during a query operation.

[1]http://tools.ietf.org/html/rfc4648

- **Result cache**
  The current version of Oracle NoSQL Database does not provide a result cache. Even if in future releases, it would not be able to cache the complete persistent object, as needed by the Gora framework. The Gora-OracleNoSQL module provides a result cache that stores in memory the last retrieved persistent object in order to provide it to Gora in a potential subsequent request. This functionality is transparent to the end-user of the module.
- **Embedded Oracle NoSQL server**
  The Gora framework requires an embedded version of the back-end NoSQL database to be available in order to automatically start it during the execution of the test suite. This way, the DataStore test methods will be able to use the specific Gora datastore module for object persistence and test their functionality. However, the Oracle NoSQL Database does not provide an embedded version of its server. The Gora-OracleNoSQL module provides such an embedded server that is started and terminated automatically during the set up and the tear down of the test suite.
- **Persistence of all the primitive datatypes and complex types**
  Each field of the data beans should be properly persisted in the back-end database. The Gora-OracleNoSQL module is able to serialise and persist all the primitive datatypes, such as int, long, char, boolean, float, etc. Additionally, it is able to serialise and persist the Avro complex types [2]: Records, Arrays, Maps, and Unions. Note that it is also able to persist nested Records and Unions with 2 or 3 types. These persistence capabilities enable the module user to define elaborate data beans that contain fields of a plethora of data types; primitive or complex ones.

5.3 System Design

5.3.1 High Level System Design

In this section, we will provide three UML diagrams whose aim is to convey to the reader a high-level overview of how the system is implemented.

5.3.1.1 Use Case Diagram

The following diagram is a UML Use Case Diagram, which illustrates the functionality available to the users (application developers) and to the Apache Gora framework. It also illustrates the system boundaries.

[2]http://avro.apache.org/docs/current/spec.html#schema_complex

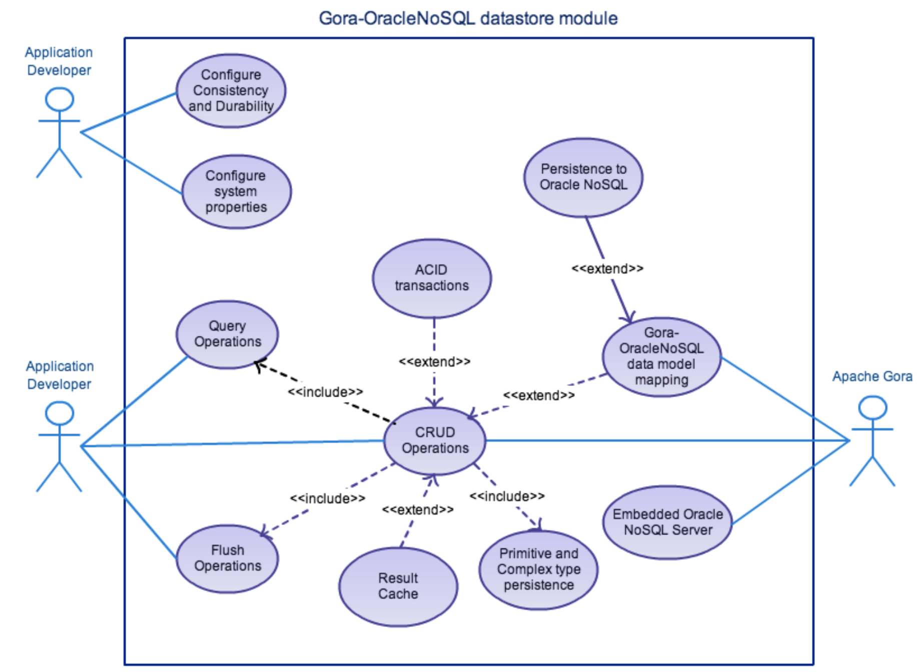

Gora-OracleNoSQL datastore module
Application Developer
Configure Consistency and Durability
Configure system properties
Persistence to Oracle NoSQL
<<extend>>
ACID transactions
Gora-OracleNoSQL data model mapping
Query Operations
<<extend>>
<<include>>
<<extend>>
Application Developer
Apache Gora
CRUD Operations
<<include>>
<<extend>>
<<include>>
Embedded Oracle NoSQL Server
Flush Operations
Result Cache
Primitive and Complex type persistence

Figure 5.1: Gora-OracleNoSQL - Use Case Diagram

5.3.1.2 Package Diagram

The following diagram is a UML Package Diagram, which depicts the system package, its main classes, and the way it integrates with the Apache Gora packages and classes.

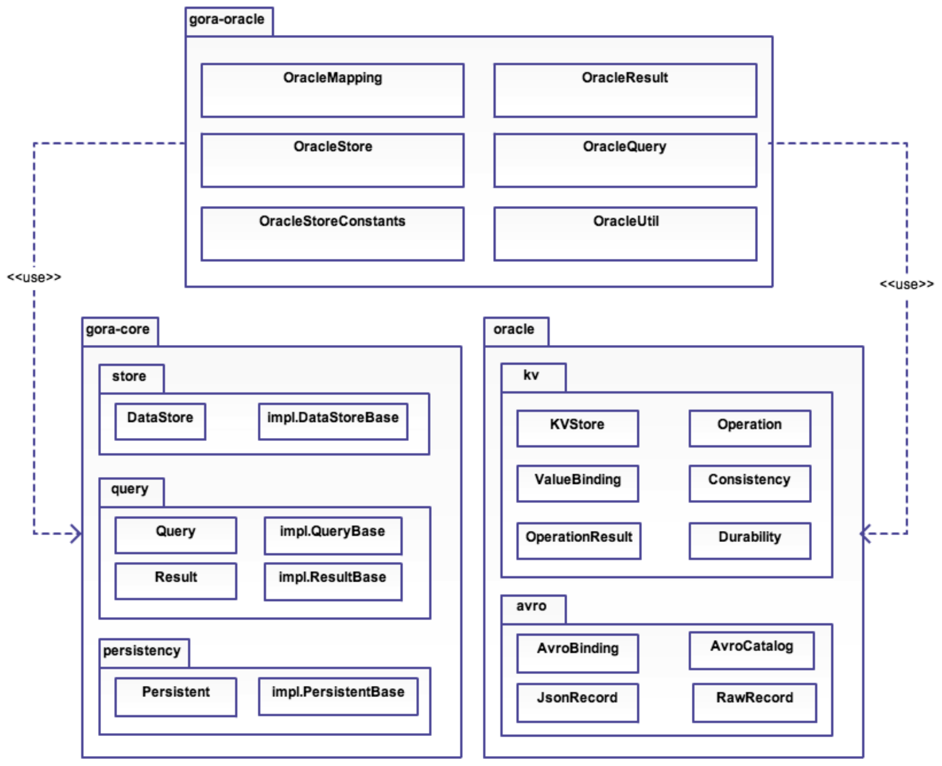


Figure 5.2: Gora-OracleNoSQL - Package Diagram

5.3.1.3 Class Diagram

The following diagram is a UML Class Diagram for the main system classes. Note that some utility classes, even though they contain valuable functionality (such as OracleUtil, OracleStoreConstants, etc), are not illustrated in the diagram because they provide only static properties, and thus no objects were created from these classes.

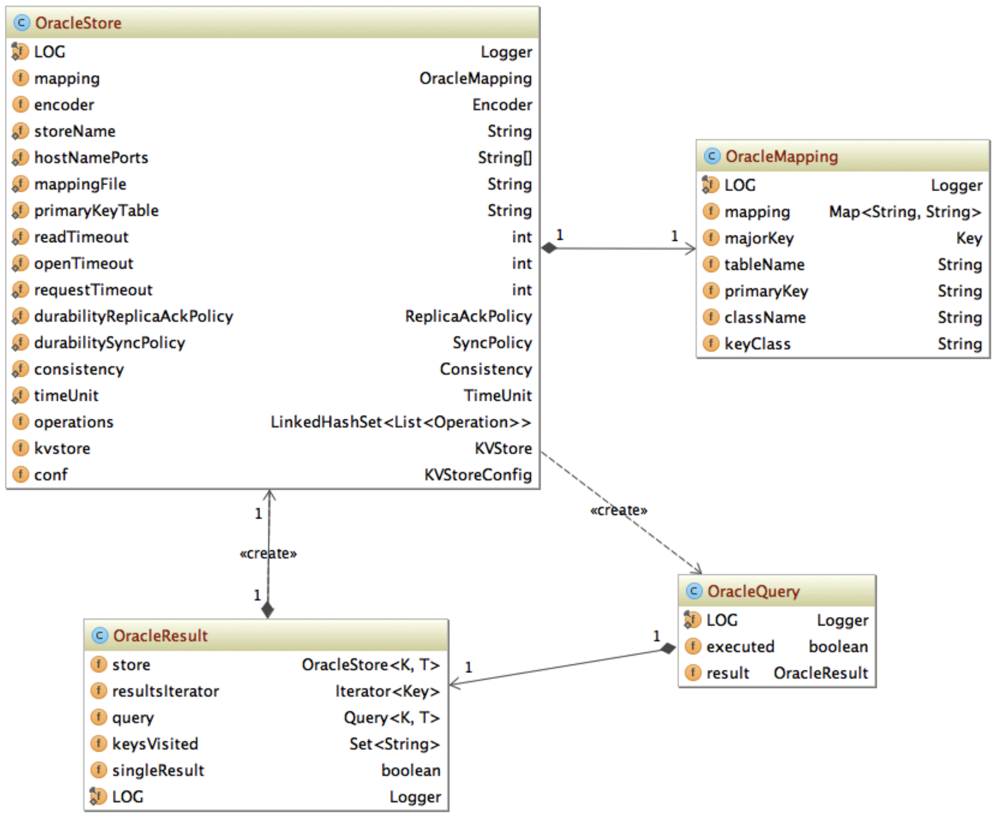


Figure 5.3: Gora-OracleNoSQL - Class Diagram

5.3.2 Low Level System Design

In this section, we will provide a brief description of the entities of the Gora-OracleNoSQL datastore module.

5.3.2.1 OracleStore

The OracleStore class provides datastore functionality to the Gora framework. It is the most important component of the Gora-OracleNoSQL datastore module and it is the component that "glues" together the functionality of all other classes of the module. It contains attributes that store the specific configuration values for the KVStore, a handle to the back-end KVStore, a data structure that accumulates the CRUD operations and a Mapping instance. The OracleStore extends the DataStoreBase class and is responsible for handling the actual object persistence of a single Gora data bean. It provides methods for initialisation, CRUD operations, schema management, and query operations. Details on the functionality of these features are presented in the section 5.4.

5.3.2.2 OracleMapping

The OracleMapping class provides mapping definitions for Oracle NoSQL. It holds information about the definition for a single table. Its main attribute is a Map<String,String> structure that maps a Gora data bean field to the key component of the Oracle NoSQL key/value pair. Additionally, it holds information about the major key that serves as a table, the name of the Gora data bean field that serves as the primary key, the class type for the persistence key, and the full class name of the Gora data bean. It is important to note that this class should be thread safe, mostly because in the future, multiple Hadoop jobs might run in parallel and this might cause synchronisation issues. To achieve thread safety, an inner builder class called **OracleMappingBuilder** is responsible for creating a single Mapping object, using simple immutability.

5.3.2.3 OracleQuery

The OracleQuery class provides Oracle NoSQL specific implementation of the Query interface. It encodes the range keys using base32hex encoding and additionally provides a Result Cache mechanism for performance optimisation. Objects of this class are used to perform queries that retrieve one or more records from the Oracle NoSQL Database. Details on the functionality of these features are presented in section 5.4.5.

5.3.2.4 OracleResult

Objects of the OracleResult class are responsible for handling Query objects and iterate through the result set that was obtained after the Read operation that was performed in the DataStore.execute() method. The main method of interest in this class is the nextInner() method, that retrieves the next key/value pair from the result set. While there are records in the result set, nextInner() stores the next record in the *persistent* attribute, which is a variable of generic type that stores the currently retrieved persistent object. When there are no more records in the result set, the methods returns false. OracleResult objects are used by the OracleQuery and the OracleStore classes.

5.3.2.5 OracleUtil

The OracleUtil class is a utility class with static methods that provide utility functionality to the other classes of the module. Most of this class's methods are related to Key functionality, such as key encoding/decoding, key creation, and primary key retrieval from a key range. These static utility methods are public in order to be available to be used by other components, if needed.

5.3.2.6 OracleStoreConstants

The OracleStoreConstants class is a wrapper class for the system constants. These constants are wrapped inside this class in order to improve code readability.

5.3.2.7 GoraOracleTestDriver

The Gora framework requires the use of an embedded back-end server in order to automate the execution of the test cases. The Oracle NoSQL Database does not provide an embedded version. The GoraOracleTestDriver contains methods that spawn a new server process, terminate the server process, create a server handle and clean up the unnecessary server files after its termination. This class is only used by the Gora test suite in order to have direct connectivity with the back-end server.

5.4 Implementation

In order for Apache Gora to be able to recognise the Gora-OracleNoSQL datastore module as a valid Gora datastore module, some aspects of its implementation had to be done in a specific way. Most notably, the new datastore classes such as the OracleStore, the OracleQuery and so on, should extend the Gora base classes and implement the required abstract methods. Also, its filesystem should follow the one indicated by Maven, its source code should be placed in a folder at the same level as the other Gora modules and the parent pom file should be modified to add the new module.

Note all this will achieve is merely to get the new module recognised as a Gora datastore module. For the new datastore to provide valid functionality, it also needs to be able to integrate with the back-end database in such a manner that Gora persistent objects can be handled properly. To achieve this, a data model should be created that will allow the persistence of the objects in a way that will successfully map the Gora data model. For this reason a data model was created for the Gora-OracleNoSQL module.

This section will clearly describe the data model of the Gora-OracleNoSQL module. We will also discuss how the functionality of its major features is achieved along with some of their implementation details. Additionally, we will present briefly the APIs that were used in this project and we will provide information about how the system was tested. Finally, we will present the assisting software that was created for this project and we will briefly discuss about their capabilities.

5.4.1 Data Model

In this section we will describe in detail how the data model of the Gora-OracleNoSQL works. We will describe the mapping between the Gora data model and the Oracle NoSQL data model. This data model is of paramount importance in this project, because it is this model that allows the object persistence in the Oracle NoSQL Database to be achieved.

Then, we will describe the structure of the mapping file, how Gora-OracleNoSQL maintains the primary keys and finally how the natural sort order of the keys is preserved.

5.4.1.1 Data model mapping

In section 2.2.3, we described the data model that the Oracle NoSQL Database uses. A clear understanding of that data model is crucial to understand the data model mapping that we created for the Gora-OracleNoSQL datastore module.

Apache Gora in its API uses some terms that do not exist in the Oracle NoSQL database: terms such as table, schema, column, and column family. Oracle NoSQL knows only about key/value pairs. However, because of the multi-component nature of the Oracle NoSQL keys, we were able to emulate a table, a column, and a column family. Following is the data model that Gora-OracleNoSQL datastore uses:

- We use the 1st component of the Major key to map the table/schema.
- We use the 2nd component of the Major key to map the persistent key.
- We use the 1st component of the Minor key to map the column/field family (if any).
- We use the 2nd component of Minor key to map the field.

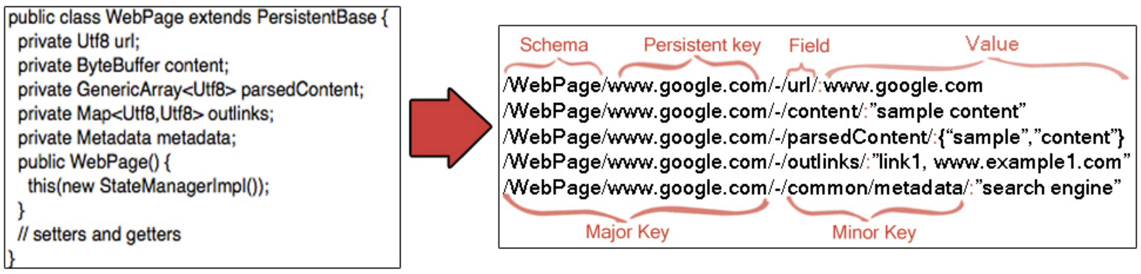
Then this maps to the following *fullKey : value* pair

/TableName/PersistentKey-FieldFamily/Field : Value

Note:

- By “table” we do not mean an actual table. It merely means that this key component would act as a container (called table in several NoSQL databases; not to be related to the relational table notion of RDBMSs). In Gora API the term schema is loosely defined. In practice, a data store can define what “schema” means for its data model. Other Gora data stores use a table as a schema. In Oracle NoSQL there is no schema nor table. Therefore, it was decided that the 1st component of the Major key would serve both as a table and a schema for the Gora-OracleNoSQL datastore.
- By “field family” we mean that this key component would serve as a container for fields that have common context. We do not mean that the Oracle NoSQL Database actually supports field/column families. This feature was added in order to be consistent with the existing Gora datastores.
- The value of each field is stored in the Value part of the key/value pair as byte array value, which is a serialised form of the field value. Gora-OracleNoSQL is responsible for serialising and deserialising the value.

Following is an illustrated example of how this data model maps a Gora data bean to Oracle NoSQL key/value pairs:



```
public class WebPage extends PersistentBase {
  private Utf8 url;
  private ByteBuffer content;
  private GenericArray<Utf8> parsedContent;
  private Map<Utf8,Utf8> outlinks;
  private Metadata metadata;
  public WebPage() {
    this(new StateManagerImpl());
  }
  // setters and getters
}
```

```
/WebPage/www.google.com/-/url/:www.google.com
/WebPage/www.google.com/-/content/:"sample content"
/WebPage/www.google.com/-/parsedContent/:{"sample","content"}
/WebPage/www.google.com/-/outlinks/:"link1, www.example1.com"
/WebPage/www.google.com/-/common/metadata/:"search engine"
```

Figure 5.4: Gora data bean to Oracle NoSQL data model mapping

Please note that this illustration is a high-level view of the mapping. The data model mapping is further elaborated in order to achieve preservation of the natural order of the keys. This is explained in detail in section 5.4.1.4.

5.4.1.2 Mapping file

An important piece that is needed in order to achieve the above-mentioned mapping is the mapping file. In Gora the mapping file is an XML file that is specific for each datastore. This means that each datastore has slightly different semantics for its mapping file compared to the other datastore’s mapping files.

The Gora-OracleNoSQL module uses a mapping file as the following:

Listing 5.3: Example of a Gora-OracleNoSQL mapping file

```
<gora-orm>
  <class name="org.apache.gora.examples.generated.Employee"
                 keyClass="java.lang.String" table="Employee">
  <primarykey name="ssn" column="ssn" />
  <field name="name" column="info/name" />
  <field name="dateOfBirth" column="info/dateOfBirth" />
  <field name="salary" column="info/salary" />
  <field name="boss" column="info/boss" />
  <field name="webpage" column="info/webpage" />
  </class>

  <class name="org.apache.gora.examples.generated.WebPage"
                keyClass="java.lang.String" table="WebPage">
  <primarykey name="url" column="url" />
  <field name="content" column="content" />
  <field name="parsedContent" column="parsedContent" />
  <field name="outlinks" column="outlinks"/>
  <field name="metadata" column="common/metadata" />
  </class>
</gora-orm>
```

As can be seen from the mapping file, the developer is able to define the name of the table, the field names of the data bean and how they should be named in Oracle NoSQL along with any possible field family if desired. An important thing to notice here is the *primary key* element. This element informs the Gora-OracleNoSQL module which field of the data bean acts as its primary key. This field should be unique among all the objects of each data bean.

5.4.1.3 Primary keys

As mentioned above, each persistent object has an associated primary key that identifies it uniquely among all the persistent objects. The primary keys are very important when multiple persistent objects need to be returned by a Query. The Query specifies a key range. The primary keys that are inside this key range should be returned when the query is executed. However, the data model that is illustrated in figure 5.4 is impractical for quick retrieval of the unique keys of the persistent objects. This is because, as shown in figure 5.4, each Gora persistent object is translated into several Oracle NoSQL key/value pairs.

In order to simplify this structure and achieve a more efficient retrieval of the unique keys of the persistent objects, we created another structure inside the Oracle NoSQL Database. This is merely another Major key component named "PrimaryKeys", that stores the persistent keys for each data bean. The full Major key that stores the primary keys of each data store has the following format:
/PrimaryKeys/TableName/-/PersistentKey :

Note that the TableName is included in the Major key path, in order to separate the persistent keys of each data bean. Also note that the persistent key is stored in the Minor key component. Finally, note that there is no need for a value; all the required information is stored in the key part of the key/value pair.

The name of the Major key component that will be used to store the primary keys is configurable by the properties file (see section 5.4.3). Its default value is "PrimaryKeys".

A practical example of usage that demonstrates the retrieval of primary keys from this data structure is the *OracleUtil.getPrimaryKeys()* method.

5.4.1.4 Preserving key natural sort order

The illustration of the figure 5.4 is a high-level view of how the mapping works. However, this mapping lacks an important feature, that we will describe now.

Gora Queries may return multiple persistent objects. In case the persistent keys are character string such as "www.google.com", "www.bbc.co.uk", then the keys of the persistent objects will be returned sorted by their natural sort order, which is the lexicographic order. However, in the case of numerical strings such as "1", "2", "10", and so on, the keys are not going to be returned sorted by their natural order, which should have been "1", "2", "10", but in the order "1", "10", "2", which is the lexicographic order and not the numeric order. For this reason, the following transformation is performed for each persistent key.

Before storing the key in Oracle NoSQL, each persistent key is encoded using the base32hex encoding. We did use this encoding specifically (and not for example the base64 or the simple base32), because, as the RFC 4648 [3] specifies:

> "One property with this alphabet, which the base64 and base32 alphabets lack, is that encoded data maintains its sort order when the encoded data is compared bit-wise."

This allows us to retrieve the keys while preserving their natural order. It must be noted here that in order for numerical string to be properly retrieved in their natural order, they should be

[3] http://tools.ietf.org/html/rfc4648

first padded with zeros so that they all have a fixed length. This task is left to the application developer, as it depends to the value types of the desired keys. However, the *OracleUtil* class provides the static utility method *padKey()* that provides String padding functionality.

Therefore, the actual key/value pairs that are stored in Oracle NoSQL are based on the data model as illustrated in figure 5.4, but the persistent keys are not plain String values. They are base32hex encoded.

5.4.2 CRUD Operations

The Gora-OracleNoSQL datastore module provides full CRUD operations. An overview of the available CRUD operations is given in section 2.3.3, that discusses the Gora API. Here, we will provide a brief overview of how each CRUD operation is implemented. Note that K is the generic type of the key class and T is the generic type of the value class (i.e. the data bean)

- **get(K key) → T**
  First retrieves a SortedMap of the key/value pairs from the database that corresponds to the given persistent key. In case no key/value pairs were returned, the method returns null. If key/value pairs are retrieved, then a new instance of the persistent object is created and its fields are populated with the retrieved values. Note that each value is deserialised according to its type. Next, the object's stateManager is cleared, in order to be able to accept future changes, if any. Finally, the new persistent object is returned.
- **get(K key, String[] fields) → T**
  Its implementation is the same as the simple get(K key) method, with the exception that only specific key/value pairs are retrieved from the database.
- **put(K key, T obj)**
  First, it checks the object's stateManager. If the stateManager is not flagged as dirty, then the put method terminates, without any further processing of the object. Otherwise, it creates a new list of operations (in order to perform all the put operations of the subsequent, multiple key/value pairs in a single transaction (see section 5.4.4)). Then it encodes the key using base32hex (as described in section 5.4.1.4), inserts the primary key as a new key/value pair under the "PrimaryKeys" Major component, and finally inserts a new key/value pair for each field of the persistent object. Two important points should be highlighted here:
  - In case a key/value pair with the same key exists in the back-end, it will silently be replaced. This effectively provides **update** semantics.
  - In case the new value of a field is empty or null, the corresponding key/value pair will be deleted from the database. This effectively provides **delete-field** semantics.
- **delete(K key)**
  To delete a persistent object based on a given key, it first creates a new query and sets its key to be equal to the given key. Then, it calls the *deleteByQuery* method (see section 2.3.3) based on this query. This effectively deletes all the key/value pairs that correspond to the given persistent key along with its primary key.
- **delete(T obj)**
  This method deletes a persistent object based on the persistent object itself. It uses the mapping to identify which field contains the primary key and extracts the value of that field. Then it works as the simple version of the *delete(K key).*

5.4.3 Properties

The Gora-OracleNoSQL module can be configured through the use of a properties file. This file should be named gora.properties file and should be located in the CLASSPATH. The only property that is mandatory is the **gora.datastore.default**. All the other properties are optional, which means that the defaults will be used for each property. The available properties, that are parsable by the Gora-OracleNoSQL module, are the following:

- **mapping.file** - Specifies the filename of the mapping file. In case this property is empty, the default is gora-oracle-mapping.xml.

- **gora.datastore.default** - Specifies the Gora datastore class that will be used for object persistence. To use the Gora-OracleNoSQL datastore module, this value should be org.apache.gora.oracle.store.Oracl
- **gora.oraclestore.storename** - Specifies the name of the KVStore. The default is kvstore.
- **gora.oraclestore.hostnameport** - Specifies the hostname and port of the node(s) that the system should connect to. The value should be one or more values of the format "hostname:port [,hostname:port]". In case of multiple hosts, a comma should be used to separate the entries, such as: localhost:5000, 192.168.0.1:5000, 192.168.0.2:5000. The default is "localhost:5000".
- **gora.oraclestore.primarykey_tablename** - This property specifies the name of the table that will store the primary keys. By table, we mean the first major component of the Oracle NoSQL key. The default is "PrimaryKeys".
- **gora.oraclestore.autocreateschema** - The autocreateschema property specifies whether the schema will be created during the initialisation or not. This merely creates the kay/value pairs, with the specific major key components that will be used to store the key/value pairs of the persistent objects and the primary keys. The default value is true.

Following are the properties that configure the levels of durability, consistency and various timeouts for the connection to the Oracle NoSQL server. The various values of these properties and their meaning can be found in the Oracle NoSQL javadoc [4].

- **gora.oraclestore.durability.syncpolicy** - The default is WRITE_NO_SYNC.
- **gora.oraclestore.durability.replicaackpolicy** - The default is SIMPLE_MAJORITY.
- **gora.oraclestore.consistency** - The default is NONE_REQUIRED.
- **gora.oraclestore.time.unit** - The default is MILLISECONDS.
- **gora.oraclestore.request.timeout** - The default is 5000.
- **gora.oraclestore.read.timeout** - The default is 30000.
- **gora.oraclestore.open.timeout** - The default is 5000.

5.4.4 ACID Transactions

One important feature of the Gora-OracleNoSQL module is that it supports ACID transactions, where available. In this section we will clarify what "where available" means, and also how this is accomplished.

The Oracle NoSQL Database provides support for ACID transactions and atomic operations [5]. This means that multiple CRUD operations will be executed as a single atomic unit. This provides ACID transaction semantics as long as all the operations are associated with records that have the same Major Key path. This way, the operations will be executed as a transaction; either all will be executed successfully or none of them.

To make use of this Oracle NoSQL feature, the Gora-OracleNoSQL accumulates all write operations (insert / update / delete) in a LinkedHashSet<List<Operation>> data structure, called *operations*. Every entry of the *operations* data structure contains a list of operations. This list of operations is specific to a single persistent object; thus, a list of key/value pairs that share the same Major Key path. The accumulated operations are executed in order, when the *DataStore.flush()* method is called. More specifically, when the *flush()* method is called, the system iterates throught the *operations* data structure and fetches every List<Operation> entry. Then, it executes the list of operations directly in the back-end, which will be processed as a single atomic unit. Finally, after the flush() method completes, the *operations* data structure will be empty in order to be clean for future operations.

5.4.5 Query operations

Another feature is the support for queries. A Query is a Gora structure that is defined in the Query interface. Its base class is called QueryBase, which provides the basic query functionality.

[4]http://docs.oracle.com/cd/NOSQL/html/javadoc/
[5]http://www.oracle.com/technetwork/database/nosqldb/overview/nosql-transactions-497227.html

Then, every Gora datastore module should provide custom query implementation according to the capabilities and the API of the back-end.

The QueryBase has 2 important attributes; the startKey and the endKey. These represent the range in which the persistent keys should be retrieved during the execution of the query. It could be an open range, which means that either the startKey will be set or the endKey. Alternatively, it could be a closed range, which means that both the startKey and the endKey are set. Queries are executed by the *DataStore.execute()* method, which returns a Result object. The Result object should be created in the execute() method of each DataStore and its contents are always based on the given Query object.

The OracleQuery extends the capabilities of the QueryBase to provide custom functionality and features. More specifically, the startKey and endKey are base32hex encoded, as this is required by the data model of Gora-OracleNoSQL module. Additionally, it provides a Result Cache mechanism.

5.4.5.1 Result Cache

This feature was implemented without being required by the Gora API. In fact, no other Gora datastore module provides a similar functionality. We decided to add this feature when we were exploring the DataStore testing suite of Gora. Several of its test cases followed the same pattern: a Query object was created and then it was executed several times throughout the method. A characteristic of this use case is that it might become overkill in case the query retrieves thousands or even millions of records. For this reason, we implemented this optimisation feature, which prohibits a query being reexecuted in the back-end, if the same query was executed in the previous call of DataStore.execute(). This is implemented by including a Result object inside the Query object along with a flag which indicates wether the query is new or whether it was previously executed. The flag is set to false when a new Query object is created and every time one of the startKey and endKey is changed.

5.4.6 Programming environment

For the implementation of the Gora-OracleNoSQL datastore module and the the assisting software 5.6, we used the Java programming language. The use of Java was required, as described in the non-functional requirements, because the API of Apache Gora is available only in Java. As an IDE we used the IntelliJ IDEA [6]. One factor for choosing to work using the IntelliJ IDE was the fact that it is a modern IDE, and much used in the industry, and having practical experience with it is a valuable skill for employability. Another factor is that its Ultimate edition was free for Apache developers and as I was working officially for the Apache Gora project, I was eligible to take advantage of the features offered by this edition.

A Test-Driven Development (TDD) approach [11] was used during the implementation. By adhering to this approach, we created the necessary JUnit test cases before we proceeded to the actual implementation of the specific method. By studying the test cases before working with the actual implementation of each method, we had a very efficient way to get a good understanding of what had to be implemented. This significantly simplified the implementation phase. Additionally, having such detailed JUnit tests helped to build a robust and a highly tested system.

For the automation of the build process and the execution of the test cases we used the Apache Maven 2 software project management tool. The use of Maven was required as Apache Gora is a multi-module project based on the structure of Maven. Thus, the parent pom file of the project should change to accommodate the new module. Also, the file system of the new module should adhere to the one required by Maven.

For version control we used Git. For a repository server we used GitHub. A public repository such as GitHub was a very convenient solution for the needs of this project. Also, learning to use GitHub, which is a highly-used Git repository in the field, provided me with valuable experience. The URL of the public repository of this project is https://github.com/maestros/gora-oraclenosql/blob/master/gora-oracle/.

[6]http://www.jetbrains.com/idea/

Finally, for maintaining a history of the issues and improvements of this project, the Jira bug tracking tool was used [7]. Jira is the official tool for bug tracking that is used by the Apache Software Foundation and it is publicly available for use by contributors.

5.4.7 APIs used

In this section, we will present the APIs that we used in our implementation of the Gora-OracleNoSQL datastore module and also give examples of how and why they where used.

- Apache Gora API
  First and foremost we made extensive use of the API of the Apache Gora framework[8]. This was not only to properly integrate with its core module, but also to make use of several utility classes and methods. The core API classes and methods that were used are explained in greater detail in sections 2.3.3 and 5.3.2. The utility classes that were used are the following:
  - AvroUtils
    This utility class provides static methods for handling some aspects of interacting with Avro schemas. Gora-OracleNoSQL uses the AvroUtils.getEnumValue() static method to get the Enum value of a serialised Enum field of a data bean.
  - IOUtils
    We used this class as it is the Gora-recommended for serialising and deserialising Avro Maps, Arrays and Records. Where possible, every Gora datastore should use IOUtils.serialize() and IOUtils.deserialize() to handle serialisation for these Avro complex types.

  Apart from the Gora utility classes that were used, the following classes where also used that provide Gora-specific exceptions for custom error handling:
  - GoraException
    We used the Gora specific exception when an I/O related exception had to be thrown. An example of usage is when an exception is thrown during the parsing of the mapping file.
  - OperationNotSupportedException
    We used this Gora specific Runtime Exception when a method is called that is not yet fully implemented or does not make sense in the context of that specific datastore. An example of usage in the Gora-OracleNoSQL datastore is the getPartitions() method. As also described in the Limitations section 7.4, Gora-OracleNoSQL does not yet provide functionality for supporting Hadoop jobs. Therefore, if the getPartitions() method is called, the OperationNotSupportedException is thrown, with an informative message.
- Oracle NoSQL API
  The second most important API that we used is of course the API of the Oracle NoSQL Database [9]. We used this API not only to use the CRUD related classes and methods of Oracle NoSQL, but also to set up the KVStore instance, handle properly the Key and Value structures, iterate the store to retrieve multiple key/value pairs, create and execute batch operations and properly handle exceptions. Section 2.2.4 clearly describes all the methods that are related to data handling in the Oracle NoSQL Database. These methods include CRUD methods, iteration-related methods and method for batch operations. In this section we will briefly present the classes that were used to set up the KV Store and handle the exceptions.
  - KVStoreConfig
    Represents the configuration parameters used to create a handle to an existing KV store. After parsing the gora.properties file, we created an instance of the KVStoreConfig based on the user-defined values of the properties file. These values include the list of node addresses and ports to connect to, the various timeouts, the consistency and durability levels. Then, we used this configuration object to get a handle of the KV store.

[7]https://www.atlassian.com/software/jira
[8]http://gora.apache.org/current/api/apidocs-0.3/
[9]docs.oracle.com/cd/NOSQL/html/javadoc/

  - KVStoreFactory
    We used this factory class to retrieve a handle to the KVStore, as was specified in the object created using the KVStoreConfig.

  The following Oracle NoSQL specific exception where used to catch the exceptions thrown during the various CRUD operations. The exact way that we handled them is described in section 5.5.

  - RequestTimeoutException
    Thrown when an operation cannot be completed because the configured timeout limit is exceeded.
  - OperationExecutionException
    Thrown when there is a failure to execute a sequence of operations.
  - DurabilityException
    Thrown during a write operation and indicates that the durability guarantee could not be met.
  - FaultException
    Indicates that an error occurred that cannot be handled in any other way but retrying the operation.

- Apache Avro API
  Apache Avro is presented in section 2.1.6.4. Due to the fact that Apache Gora extensively uses Apache Avro for data bean definition and serialisation, we also had to use the Avro API [10] to properly read and write the Avro schemas. For example, Avro uses the Utf8 datatype to store character strings instead of the Java String datatype. Another example of its usage is the class SpecificDatumReader and SpecificDatumWriter used to read and write Union complex types. Finally, the BinaryEncoder and BinaryDecoder are used to serialise and deserialise Avro values into binary-format.
- JDOM API
  JDOM's SAXBuilder [11] was used to properly parse the XML mapping file and extract the values from the specific mapping elements.
- SLF4J & Log4j API
  We used SLF4J [12] and Log4J [13] to efficiently use multilevelled logging of the generated log messages.
- Base32Hex
  As described in the section 5.4.1, there is an inherent problem between the keys of the data models of Gora and Oracle NoSQL. In order to preserve the natural sort order of the Gora keys and being able to retrieve multiple keys in their natural sort order, we used the base32hex encoding, which, according to the RFC 4648, maintains the sort order of the encoded data. We used the com.buck.common.codec.Base32Hex [14], licensed under the Apache License, Version 2.0, as the implementation of the base32hex encoding.

5.5 Testing

In this project, we used a Test-driven development methodology. For each feature that we wanted to implement, such as CRUD operations, Query operations, and so on, we first wrote an test case meant to test and validate the new functionality. Of course at first this test case would fail, but it was used to drive the implementation of the actual functionality. Next, we added a first version of the implementation and validated its functionality against the test case. Then, we refactored the implementation until the test case passed successfully.

The Apache Gora has an existing testing suite that provides plenty of test cases to validate the datastore-specific implementation of each API method (**validation**). Also, by having

[10]https://avro.apache.org/docs/current/api/java/
[11]http://www.jdom.org/docs/apidocs/org/jdom2/input/SAXBuilder.html
[12]http://slf4j.org/apidocs/org/slf4j/Logger.html
[13]http://logging.apache.org/log4j/2.x/
[14]http://grepcode.com/file/repo1.maven.org/maven2/com.github.rbuck/java-codecs/1.0.1/com/buck/common/codec/Base32Hex.java

successful test cases, we can verify that the system conforms to the functional requirements as specified in section 4.2.1 (**verification**). Most of the Gora-OracleNoSQL JUnit tests override the DataStoreTestBase test cases, which are provided by the Gora API and are mandatory to be successfully passed by every datastore module. Also, several methods for asserting the final result of the test cases had to be created. These assertion methods verify the **consistency** of the persistent data directly from the Oracle NoSQL Database. Additionally, for the extended functionality that the Gora-OracleNoSQL module provides, specific test cases were created. Finally, some additional test cases were created to further test the robustness and the capabilities of the system.

Following is a list of the test cases that where created and the assertion methods. For each one, a brief description is added.

- testAutoCreateSchema - Tests and asserts that the autocreateschema property works as expected.
- testTruncateSchema - Tests and asserts that schema truncation works as expected.
- testDeleteSchema - Tests and asserts that the DataStore is able to delete a schema.
- testSchemaExists - Tests the existence of a schema, by creating a new schema.
- assertSchemaExists - Asserts that the schema created in the testSchemaExists, exists.
- testPut - Tests that the put operation works as expected by persisting a new object.
- assertPut - Asserts that the new object, created in the testPut, is persisted as expected and retrieved properly.
- testPutNested - Tests and asserts that an object that contains a nested object is persisted as expected.
- testPutArray - Tests that an object that contains an array field is persisted.
- assertPutArray - Asserts that the object created in the testPutArray is persisted as expected and retrieved properly.
- testPutBytes - Tests that an object that contains a byte array or a character string field is persisted.
- assertPutBytes - Asserts that the object created in the testPutBytes is persisted as expected and retrieved properly.
- testPutMap - Tests that an object that contains a Map field is persisted.
- assertPutMap - Asserts that the object created in the testPutMap is persisted as expected and retrieved properly.
- testUpdate - Tests and asserts that an update operation works as expected.
- testEmptyUpdate - Tests and asserts that an update operation works as expected, if the new value is null.
- testGet - Tests that a get operation works as expected, by persisting a new object and asserting that it is retrieved properly.
- testGetRecursive - Tests and asserts that an object with a nested recursive object is persisted as expected and retrieved properly.
- testGetDoubleRecursive- Tests and asserts that an object with a double nested recursive object is persisted as expected and retrieved properly.
- testGetNested - Tests and asserts that an object that contains a nested object (not recursive) is persisted as expected and retrieved properly.
- testGet3UnionField - Tests and asserts that an object with a 3-types union field is persisted as expected and retrieved properly.
- testGetWithFields- Tests that a get operation with specific fields works as expected, by persisting a new object and asserting that it only the specific fields are retrieved.
- testGetWebPage - Tests that a get operation works as expected, by persisting a new WebPage object and asserting that it is retrieved properly.
- testGetWebPageDefaultFields- Tests that a get operation works as expected, by persisting a new WebPage object and asserting that all its fields are retrieved properly.
- testGetNonExisting - Tests that a get operation works as expected when trying to retrieve a non-existing persistent key.
- testDelete - Tests and asserts that a delete operation works as expected.
- testDeleteByQuery - Tests and asserts that multiple persistent objects are deleted based on specific queries.

- testDeleteByQueryFields - Tests and asserts that only specific fields of the identified persistent objects are deleted base on specific queries.
- testDeletePersistentObject - Tests and asserts that persistent objects can be deleted given the same persistent objects and not only given their keys.
- assertTopLevelUnions - Asserts that when writing a top level union ['null','type'] the value is written in raw format.
- assertTopLevelUnionsNull - Asserts that when writing a top level union ['null','type'] the null value is treated properly.

The following test cases verify that the various combinations of query setups work as expected by retrieving the expected number of objects according to the each query setup.

- testQuery
- testQueryStartKey
- testQueryEndKey
- testQueryKeyRange
- testQueryWebPageSingleKey
- testQueryWebPageSingleKeyDefaultFields
- testQueryWebPageQueryEmptyResults

We should mention an issue with testDeleteByQueryFields method. This test method test an important feature of the datastore: the ability to delete specific fields from multiple persistent objects using a Query operation. This feature works **correctly** in the Gora-OracleNoSQL datastore. However, the test method testDeleteByQueryFields is defined by Gora's DataStoreTestBase and it has a problem with unambiguity concerning the query range criteria. These criteria are interpreted differently in some Gora datastore modules. This is a known Gora bug [15], which is still not resolved. We had to ignore this test method until the Gora-66 issue is resolved. However, the functionality of this feature works correctly in the Gora-OracleNoSQL and this can be demonstrated by the LogManager (see section 5.6).

The above test cases perform a thorough **Unit Testing** of the system. After the completion of the implementation of every method, we run the whole test suite both to perform **Regression Testing** and as a way to validate the new functionality.

As we will discuss in the next section, we used an assisting application, called LogManager, to test and demonstrate the functionality of the Gora-OracleNoSQL datastore module. Through the use of this software we performed extensive **simulation** before the submission of the project. This effectively contributed to a successful **System Testing**.

5.6 Assisting software

The Gora-OracleNoSQL datastore module is able to work successfully without the use of any assisting software, apart from its dependencies of course. However, for this project we used two additional applications to assist us during the demonstration of the functionality of the Gora-OracleNoSQL datastore module. The first one is the OracleTerminal and the second one is the LogManager.

The Apache Gora framework is a framework that is used as an abstraction storage layer for the front-end applications. Therefore, as a back-end system, its functionality cannot be demonstrated on its own. For this reason, a sample front-end application is needed that will make use of the functionality provided by the Gora framework. The LogManager is the official sample application that is used by the Apache Gora community to demonstrate the functionality of the Gora framework. The **LogManager** is a sample application that manages web server logs. It is able to parse any web server log file, given that the log format is the Combined Log Format [16]. The LogManager is able to parse web server log files and storing the logs, deleting some lines and querying.

[15] https://issues.apache.org/jira/browse/GORA-66
[16] http://httpd.apache.org/docs/current/logs.html

We considered using the LogManager application in order to be consistent with the demonstrating approach of the rest of the Gora community. However, we did not use the LogManager exactly as it is provided by the Gora community. We **customised** its functionality for two reasons. First, because of the limitation of the Gora-OracleNoSQL module that supports persistent keys only of type String. The LogManager uses keys of type Long. Therefore, we transformed the Long keys into zero padded Strings with a fixed length of 4 characters. Second, we added a new read capability that extends its current capabilities. This is the *-get <lineNum> [<field list>]* option, which we believed it is a feature that was important to be demonstrated.

The available options of the LogManager are the following:

- **-parse <input_log_file>**
  This option parses the input file that contains the web server logs. Each entry of the file is parsed into a separate object that is persisted in the database. After the completion of this operation, all the entries of the log file will have been parsed into separate objects and will have been persisted into the database. The key for each persistent object will be the line number of each entry of the log file.
- **-get <lineNum> [<field list>]**
  This option allows the user to retrieve the persistent object that is associated with the given key. As described above, the key for each persistent object is its line number in the log file. The retrieved persistent object will be displayed on the screen. This calls the toString() method of each persistent object, which displays all the object fields and their values. Note that this option, in its simple form, retrieves the values of **all** the fields of the object. However, there might be some use cases in which this behaviour is not needed. For this reason, Gora and the Gora-OracleNoSQL datastore provide an overloaded method that retrieves only specific fields for the persistent object of the given key. Therefore, the optional <field_list> parameter retrieves specific fields for the given persistent object. An example call of this option would be the following:
  LogManager -get 15 ip,url
  The above command will retrieve the persistent object with the key 15 and only the fields ip and url will have values retrieved from the database. The rest of the fields will be empty or zero.
- **-query <lineNum>**
  This option creates a Gora Query and searches the back-end database for a single persistent key. If it is found, the associated persistent object is returned. If not, no object is returned and an informative message is displayed to the user.
- **-query <startLineNum> <endLineNum>**
  This option creates a Gora Query and searches the back-end database for persistent objects that fall inside the given key range. If there are stored keys inside the given range, the associated persistent objects are returned. If not, no object is returned and an informative message is displayed to the user. This also demonstrates that the retrieved objects are returned in a sorted order, based on the natural sort order of their keys (see section 5.4.1.4).
- **-delete <lineNum>**
  This option deletes the persistent object from the database that is associated with the given key (lineNum). Note that this is translated to multiple delete operations that will be executed in the back-end database, because a persistent object uses several key/value pairs for persisting all its fields. Additionally, the primary key of this object will be deleted, which is stored under the "PrimaryKeys" Major component.
- **-deleteByQuery <startLineNum> <endLineNum>**
  This option creates a new Gora Query that specifies a key range based on the given keys (startLineNum, endLineNum). It then uses this query to perform a deleteByQuery operation, which deletes all the persistent objects whose keys are inside the query range. Additionally, the primary keys of the deleted persistent object will be deleted, which are stored under the "PrimaryKeys" Major component.

The Oracle NoSQL Database does not provide a terminal application, like other databases do. For example the Oracle RDBMS uses the SQL*Plus utility that allows the developer to submit ad-hoc queries directly to the database. Since there was no such utility for the Oracle NoSQL Database, we **created** one in order to be able to validate the results of the Gora-OracleNoSQL operations directly from the database. We named this assisting software **OracleTerminal**. Its purpose is to allow the developer to connect directly to the Oracle NoSQL Database and access in an ad-hoc fashion. However, for the purposes of the demonstration, there was no need to allow the developer to submit freely any kind of ad-hoc operation. Therefore, we restricted the capabilities of this utility by creating specific options. Also, the OracleTerminal has knowledge of the Gora-OracleNoSQL data model. This means that it is aware of what a key of a persistent object is and what a primary key is.

The available options of the OracleTerminal are the following:

- **-dumpDB**
  With this option, the OracleTerminal will display on the screen all the key/value pairs that are currently stored in the Oracle NoSQL Database. This option is useful when one wants to validate that the database contains data and it is not empty or wants to visually validate that the structure of the stored key/value pairs is correct. We used this option after the parse operation of the LogManager.

  The format of the output of each key/value pair is the following:
  MajorKey-MinorKey : Value
  Note that the MinorKey and/or the Value parts are optional. This means that there might be kay/value pairs that do not have a MinorKey and/or a Value. This is valid based on the Gora-OracleNoSQL data model. Also, note that the fields of the persistent objects whose type is a complex data structures are serialised in order to be stored in the database. Therefore, because of the fact that the OracleTerminal does not perform deserialisation, these serialised values are displayed directly on the screen. This has the side effect of displaying several non-printable characters. However, as explained, this is an expected behaviour.
- **-get <key>**
  This option allows the user to get all the key/value pairs that are related to a specific persistent key. The parameter of this option (key) refers to the persistent key (the key of the persistent object).
- **-truncate**
  This option deletes all the stored key/value pairs from the database. We provided this option in order to have a clean database before the demonstration.
- **-countKeys [<PrimaryKeys>]**
  This option displays and counts the key/value pairs that are stored in the database. Note that a Gora persistent object may be composed of several key/value pairs. An optional parameter is the "PrimaryKeys" keyword, that counts only the stored primary keys; the key/value pairs that are stored under the "PrimaryKeys" Major component.

Note that the OracleTerminal utility has no capabilities for write operations. This is because, as we mentioned above, the purpose of this utility is to validate the results of the operations performed by the Gora-OracleNoSQL module. Therefore, there is no need to add write capabilities to this utility in order to successfully validate the results.

Chapter 6

Apache Gora contribution

6.1 Overview

A secondary objective of this project was to contribute to the Apache Gora community, not only by implementing the Gora-OracleNoSQL module, but also by improving the framework in general. This work was performed not only because this project took part in the Google Summer of Code 2013 event, but also because it was a convenient way of getting to know the implementation details of the Gora framework. For this reason, I monitored daily the Jira bug tracking tool and communicated with the Gora community about the unresolved bugs of the project. I must highlight here the fact that the bugs that I am referring here are not bugs of the Gora-OracleNoSQL datastore but bugs of other modules of the Gora framework. Within the time frame of this project I identified 8 new bugs and improvements and I provided patches for a total of 15 issues. It may be noted that I provided patches that affect **all** modules of the Gora framework, including all its supported data store modules. This gave me insights and valuable knowledge of the whole framework that enabled me to avoid some mistakes of other datastore modules and also to utilise some Gora API functionality that otherwise would have been unknown to me because of its limited documentation. For these reasons, this contribution proved to be very valuable for the success of the main objective of this project.

6.2 Issues resolved

The following will describe the most important issues that have been resolved during the time frame of this project. The full details of each of these issues are also available online in the Jira application that Apache Gora maintains[1].

[1]https://issues.apache.org/jira/browse/GORA

GORA-229: Use @Ignore for unimplemented test functionality
Jira issue URL: https://issues.apache.org/jira/browse/GORA-229
This issue is about the use of JUnit 4 features in the test cases, which is essentially an improvement for the testing suite of the project. More specifically, it involves refactoring all the Java classes of Apache Gora to abandon the deprecated junit.framework.* and use the recommended static org.junit.Assert.*. Additionally, it involves making use of the @Ignore annotation for the test methods that do not have proper implementation yet and should be skipped. This was a very big patch that affected all the modules and data stores of the project. The patch has already been committed to the project.

GORA-231: Provide better error handling in AccumuloStore.readMapping
Jira issue URL: https://issues.apache.org/jira/browse/GORA-231
Each Gora datastore has to read the mapping file and parse it accordingly. However, there is no check and proper error handling in case the mapping file does not exist. The goal was to create a patch that would provide a consistent error handling among all Gora datastore. For this reason I modified the DataStoreFactory.getMappingFile() which is used by all datastores. It now throws an *IOException* if the mapping file does not exist in the path. The patch was submitted but has not yet been committed at the time of writing this report. **GORA-232: DataStoreTestBase functionality delegation to DataStoreTestUtil**
Jira issue URL: https://issues.apache.org/jira/browse/GORA-232
This issue was an improvement of the implementation of the testing suite. The pattern that the test suite follows demands that alDataStoreTestUtil should be responsible for the actual testing. However, some test cases defined in DataStoreTestBase contained an actual test code implementation. The patch that I contributed delegated all the testing functionality to DataStoreTestUtil, where appropriate. The patch has already been committed to the project.

GORA-243: Properly escaping spaces of GORA_HOME in bin/gora
This bug was identified by me. The problem was that when the user executed "bin/gora" from a path that contained one or more spaces, it failed with the following exception:
Exception in thread "main" java.lang.NoClassDefFoundError: org/apache/Avro/Schema
The cause was the mishandling of spaces when building the CLASSPATH environmental variable in bin/gora. The solution was to save the IFS (Internal Field Separator) variable into a temporary variable, the build the CLASSPATH environmental variable and then restore the original value of IFS from the temporary variable. The patch has already been committed to the project.
Jira issue URL: https://issues.apache.org/jira/browse/GORA-243

GORA-258: writeCapacUnits gets value from wrong attribute
This bug was identified me and involves a variable that reads a value from a wrong property in the Gora-DynamoDB module. More specifically, I spotted that the variable writeCapacUnits was wrongly reading its value from the element readCapacUnits during the xml parsing of the mapping file. The patch has already been accepted and committed.
Jira issue URL: https://issues.apache.org/jira/browse/GORA-258

GORA-259: Removal of the main methods from the test case classes
This improvement was identified by me. I spotted that several JUnit test case classes contained main methods for triggering some test cases, which is a non-recommended practice. After I received positive feedback from the community, I proceeded to create the patch, which has already been committed.
Jira issue URL: https://issues.apache.org/jira/browse/GORA-259

GORA-264: Make generated data beans more java doc friendly
This issue is an improvement of the GoraCompiler. The goal of this improvement is to make the GoraCompiler add static and dynamic javadocs to the generated data beans. In order to make the javadocs as dynamic as possible, I extended the compiler's parsing capabilities of the json Avro schemas. More specifically, the GoraCompiler now extracts information from the "doc" element of the Avro schema in order to dynamically construct the javadoc for the generated data bean. This way, the generated data beans provide helpful comments and are much more readable and maintainable. A sample data bean generated with this patch can be seen in 2.3.2. The patch has already been accepted and committed.
Jira issue URL: https://issues.apache.org/jira/browse/GORA-264

GORA-265: Support for dynamic file extensions when traversing a directory
Until version 0.3 of Gora, the GoraCompiler had 2 ways of finding the Avro schemes that it would compile. The first one was to directly specify the filename of the Avro scheme and the second one was to to specify a directory. In the later case the GoraCompiler would compile all the Avro schemes whose filenames have the default extension of .avsc. I proposed this improvement that allows the user to specify dynamically the extensions of the files of a specific directory. This way the user has much greater flexibility when needing to compile multiple Avro schemes. The patch has already been accepted and committed.
Jira issue URL: https://issues.apache.org/jira/browse/GORA-265

GORA-268: Make GoraCompiler the main manifest attribute in gora-core
Normally, the GoraCompiler should be invoked by using the bin/gora executable. However, the GoraCompiler being a part of the gora-core module could be invoked by the maven-generated jar file. This functionality was not implemented yet. I submitted this patch that makes the GoraCompiler the mainClass in the jar's manifest. Therefore, a user is able now to simply invoke the GoraCompiler directly from java, as follows:
java -jar gora-core/target/gora-core-0.4-SNAPSHOT.jar.
This provides greater flexibility in invoking the GoraCompiler and also allows the GoraCompiler to be uncoupled from the rest of the framework.
Jira issue URL: https://issues.apache.org/jira/browse/GORA-268

6.3 Contribution Statistics

The following pie charts are provided to illustrate the amount of work that I performed for this project. The pie charts were generated by the Jira bug tracking tool that Apache Gora maintains and are also available online[2].

[2]https://issues.apache.org/jira/browse/GORA

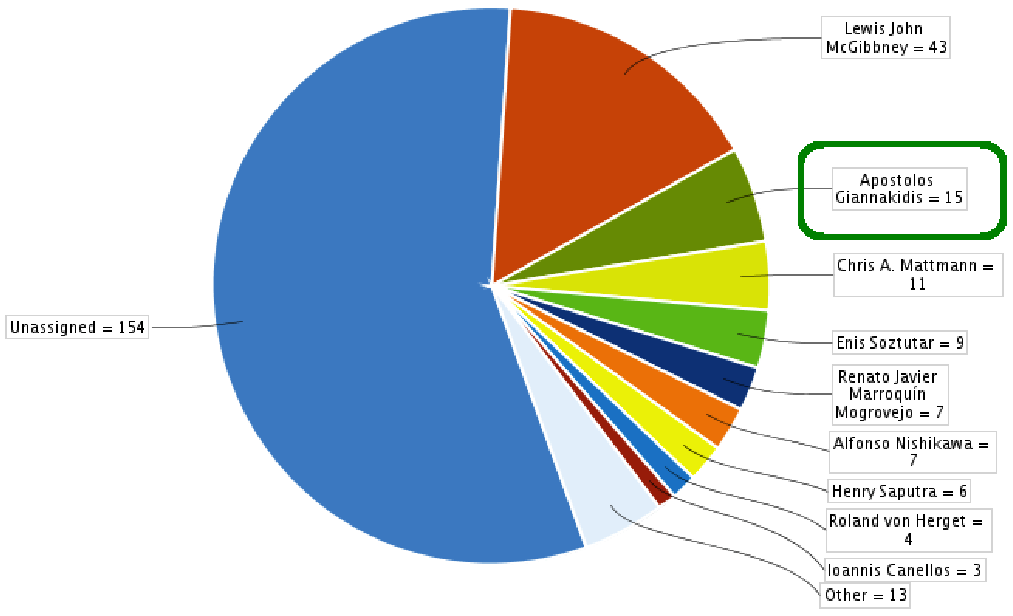


Figure 6.1: Number of issues by assignee

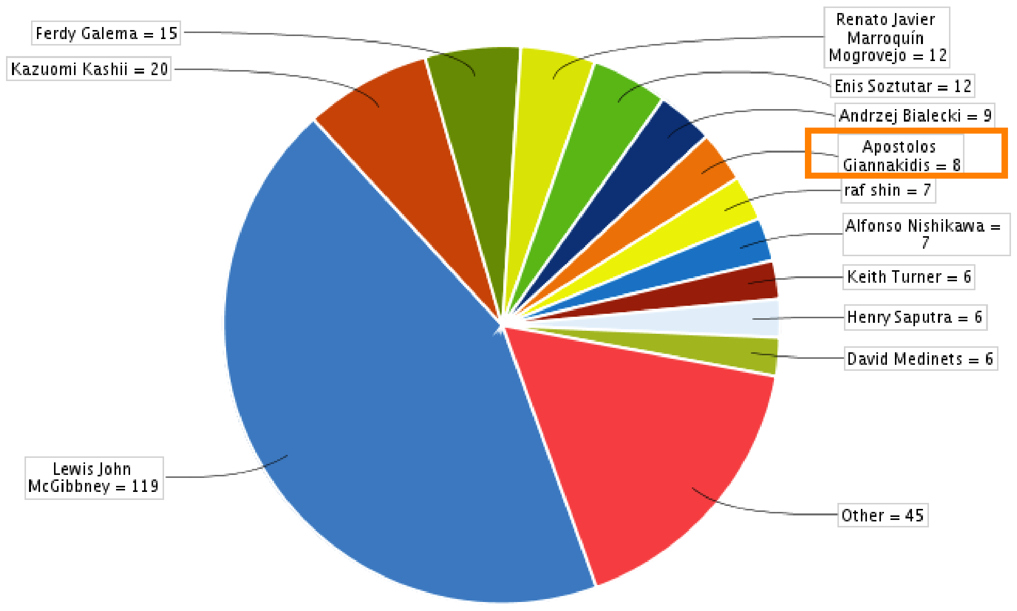


Figure 6.2: Number of issues by reporter

Note: Since this project will continue to be developed and maintained, these charts will change. The Jira URL for the most current statistics is: https://issues.apache.org/jira/browse/GORA. The pie charts can be dynamically generated by clicking on the “Pie Chart Report” option.

Chapter 7

Evaluation

This chapter will evaluate the project. It presents the learning outcomes and the most important challenges faced, and describes the achievements of the project and its current limitations. It evaluates the delivered product and the process followed to achieve its completion.

7.1 Learning outcomes

My main objective right from the outset of this project was to gain as much knowledge as possible, not only of a new field, but also of new technologies and tools. It is my belief that I achieved these learning objectives. More specifically, I have gained the following knowledge:

- Familiarity with the Big Data field
- Various NoSQL technologies and concepts:
  - Schemaless data modelling
  - Aggregate oriented databases
- How an object persistence framework achieves its goals:
  - How Object-to-Datastore mapping works
  - How to create a database abstraction
- Expertise in 3 software systems and their APIs:
  - Oracle NoSQL database
  - Apache Gora
  - Apache Avro
- Experience in the use of the following Software Engineering tools:
  - Git (version control)
  - Maven (build automation)
  - Jira (bug tracking)
  - IntelliJ IDEA (Integrated Development Environment)
- How Open Source projects are generally structured and work
  - How the Apache Software Foundation works
- Experience in working in an Open Source project
- How to organise the activities of a software development project of a complex system and properly prioritise them.
- How to find alternative solutions to issues that block the completion of a project.

7.2 Challenges

This project started with some known risks that proved to be significant challenges. Additionally, some unexpected challenges emerged along the way. All these challenges are described in detail in this section.

- Before I started the project, I had virtually zero previous knowledge in the field of NoSQL and Big Data. This was actually one of my motivations for working on a project on this field, as I wanted to get to know this very promising field of computer science. However, this total lack of knowledge was a big challenge to overcome, because in a strict timeframe I not only had to learn on my own the concepts and the fundamentals of the various NoSQL technologies, but also to learn how the Apache Gora framework and the Oracle NoSQL work, and then to analyse, design, implement and test a new data store module for this framework, which was also unknown to me previously.
- As both software systems are very new, there are no books on Apache Gora and Oracle NoSQL that could provide me with the information I needed to build up the skills necessary to the construction of the new datastore module. My main information resources were the javadocs, some blog posts and the community of each project.
- Apache Gora is a very new framework. The version 0.3 was released a few days before I started this project. Because of this, and also because it is an Open Source project, there was limited or inconsistent documentation, in some cases. For example, there is no guide on how to create a new data store module. Therefore, I had to study the code of the other data store modules and identify on my own what has to be done. This was very time-consuming and confusing because the implementation of these data stores are not consistent between each other. Each datastore is implemented in its own way, relatively different from the others. Additionally, the Gora API is not very clear in several cases, which caused significant delays while I clarified the confusion.
- There is no Oracle NoSQL database embedded version. This limitation caused significant delays until an approach for resolution was agreed. Several alternative solutions were tried, in an attempt to automate the initialisation and termination of the embedded version of the server, but it was proved that the only trustworthy solution was to manually create the embedded version of the server. The embedded version is needed by the GoraOracleTestDriver which is used by the test case suite, so that the datastore test cases can be executed without having to manually start the NoSQL database.
- There is a significant limitation of the Gora-OracleNoSQL datastore module regarding license version incompatibility. The limitation is described in detail in section 7.4. This incompatibility prohibits the distribution of the module in the Gora release. The significance of this should not be overlooked, as it invalidates my motivation to contribute to the Open Source community. However, since this project proposal was already accepted in the Google Summer of Code 2013, and since the Oracle NoSQL development and management team informed us that in the future, they might release their client API in a compatible license, I decided to move on with the completion of the project.

7.3 Achievements

Despite the challenges that are described in 7.2, the project was successfully completed and delivered on time. The following list describes the achievements of this 3-month project:

- Embarked on a NoSQL project regardless of the risk inherent in my virtually **zero previous knowledge** on the NoSQL field and successfully **completed the project on time**.
- Because of this zero previous knowledge, I had to allocate significant time to study the theory behind the vast field of the various NoSQL technologies, before starting the design and the implementation of the practical deliverable of this project. This introduced a **big risk** for the successful completion of the practical deliverable because of the little time that was left. However, despite the risk, I successfully gained significant theoretical knowledge of the various NoSQL technologies and completed the implementation of the Gora-OracleNoSQL data store module.

- Successfully researched, analysed, designed, implemented and fully documented a NoSQL software system within a **strict timeframe**.
- Understood the internals of a **framework previously unknown to me** to such a degree that I was able to improve the quality of its core functionality and of several of its modules by solving some of its open issues and by identifying some new.
- Contributed new features and improvements, and fixed bugs, on a framework that has thousands of deployments.
- The delivered software system not only incorporated almost all the features and functionality as they were described in the project proposal but also **delivered added functionality** that was not initially planned.
- I am proud to have successfully contributed to the Open Source community and been nominated for promotion to **Apache Project Management Committee (PMC) member** and Commiter status.

7.4 Limitations

At the moment of writing this report, the delivered Gora-OracleNoSQL datastore module has some limitations that have not been resolved yet. As we will see, some of them were because of lack of time, others because of legal issues and others because of unclear specifications due to the lack of proper documentation of Apache Gora. Following, the current limitations of the module are described in detail.

Even though Apache Gora has extensive support for Hadoop, the Gora-OracleNoSQL module does not provide such functionality as it is required by Gora's API. More specifically, Gora API requires its datastore modules to implement the *getPartitions(Query<K, T> query)* method. This is one of the few datastore-specific methods which eventually involves and utilises Hadoop MapReduce computation for executing Gora jobs on data within the back-end store. Because of my limited knowledge of Hadoop and because of lack of time, the implementation of this method has not been completed yet.

After the initiation of the project, we realised that there is a license incompatibility problem that prohibited distribution of the module in the next release of Apache Gora. More specifically, the Gora-OracleNoSQL module depends on the kvclient.jar of the Oracle NoSQL database. This jar is the driver that provides the necessary Oracle NoSQL API that Apache Gora must use. The fact that the license of Oracle NoSQL is GNU AGPLv3 and the license of Apache Gora is Apache License 2.0 creates a license incompatibility issue that forbids us to incorporate this jar file in the distribution of Apache Gora. We contacted the Oracle NoSQL development and management team and they informed us that they will work on a possible solution. However, this issue has yet to be resolved by Oracle. Therefore, the users that would want to use the Gora-OracleNoSQL module will have to download it separately from the main Gora distribution. The user will have to be responsible for downloading the Oracle NoSQL database and installing the kvclient.jar in the proper directory of Gora's file system.

Apache Gora has an extensive suite of test cases that validate the proper functionality of a data store module. However, even though the Gora-OracleNoSQL module successfully passed all the test cases, a limitation was identified a few days before this report was written. More specifically, in Gora the key for a persistent object could be of any data type. However, the test cases test only the case of String keys. This is very consistent with the data model of the Oracle NoSQL, as the data types of its keys are a series of strings. This limitation was identified when we tried to use the LogManager demonstration utility [1], which uses Long as the persistent key. Because of this issue in the test cases, a false impression was created that persistent keys should only be of type String. The Gora-OracleNoSQL datastore module was created based on this false impression and therefore its support for keys is limited to String.

[1] http://gora.apache.org/current/tutorial.html

7.5 Improvements and future work

Apache Gora has a great potential to become the standard in Big Data persistence. Also, I believe that this project provides important value to the framework and I will continue to maintain and improve it. Additionally, as this project is released in the Open Source community, other developers can contribute and provide improvements and new features.

The most important improvements that should be addressed in the immediate future are the following:

- The limitation for supporting Apache Hadoop should be addressed by implementing the necessary Gora API methods, such as the *getPartitions()*. Based on discussions with the Gora community, this is considered an advanced feature that will add more value to the Gora-OracleNoSQL module. Oracle NoSQL by itself integrates with Apache Hadoop, which means it is feasible to address this limitation.
- The Gora-OracleNoSQL module currently supports persistent keys only of type String. This is a significant limitation that should be addressed. Moreover, the test case suite should change and incorporate tests that will validate that all the datastore modules support keys of other types apart from String.
- If and when Oracle releases the Oracle NoSQL client API under an Apache License 2.0, the Gora-OracleNoSQL module should change in order to incorporate the necessary jar file in its path. This way the module will be distributed in the official Gora release.
- The Gora-OracleNoSQL module should integrate with the GoraCI test suite which will allow us to validate that the module can operate with a performance level similar to other modules while ensuring that data is not lost at scale. This test suite is called continuous ingest [2].

[2]https://github.com/keith-turner/goraci

Chapter 8

Project Management

8.1 Project distribution

This project was organised in 7 phases. Following is a high-level overview of the key phases of the project and the main tasks of each one. Please note that we did not follow a waterfall methodology. Therefore, some activities of these phases overlap with some others during this project.

Phase 1: Initiation and planning phase
Since I had no knowledge of Oracle NoSQL and of Apache Gora, the learning curve was substantial, as I had to learn and fully understand both systems. Following are the major tasks that I performed during the 1st phase of the project:

I studied their documentation, functionality and features. I studied and reviewed their APIs and also study the source code of both products to understand their inner functionality. It was important to identify areas of key importance of the APIs. More specifically: how specific API calls of Oracle NoSQL could be properly mapped to Apache Gora. Proper handling of CRUD operations of multi-component Oracle NoSQL keys. Special care had be taken for proper handling of CRUD operations of multiple key-value pairs within an entity (same major key). Oracle NoSQL's unordered scan API had also be studied and handled properly during get() and flash() operations. Oracle NoSQL's variable consistency for read operations and varying degrees of durability for update operations had to be studied. In this phase, I also deployed both systems and created a sample database in Oracle NoSQL. Additionally, I investigated existing solutions that integrate with Oracle NoSQL. Then, I studied the source code of other Gora modules to understand how they achieve their integration with their datastores. I studied the Oracle NoSQL data model in order to determine how its data model influences Gora's internals and how a mapping could be implemented. Finally, I identified dependencies and how they could be handled in this project.

Phase 2: Design phase
In this phase, after I had a clear understanding of the architecture and the details of both Gora and Oracle NoSQL, I created a design in UML (package diagram) to communicate to my supervisor and to the Gora community the architecture of the Gora-OracleNoSQL module that I would implement. It was agreed that there was no need to elaborate much in the UML diagrams at that point but to focus more on the actual implementation of the datastore module. More UML diagrams were created in later phases of the project.

Phase 3: Implementation phase
For the implementation, I adopted a Test Driven Development approach. Apache Gora had a ready made test case suite that I had to use. However, I also created some JUnit tests for the Oracle NoSQL datastore for the extra functionality that I wanted to add. Of course, the tests failed at first, but they guided the actual implementation of the datastore. Having such detailed JUnit tests helped in building a robust and a highly tested system. For the automation of the build process and the execution of the test cases, I used the *Apache Maven 2* software project management tool. For the version control of the project I used *Git* and I used a repository in *GitHub*. I considered it very important to commit very frequently in order to have my code reviewed by my supervisor and to receive early feedback.

Phase 4: System testing and bug fixing phase
In this phase, I tested the whole system and ran real simulations of the module in order to identify possible bugs. Any bug that was identified was logged in Gora's Jira but tracking tool and then fixed.

Phase 5: Code clean-up phase
In this phase, I performed a source code clean-up and I fully documented it with proper Javadoc and inline comments. Having proper comments and javadocs is essential for this project, first because it has to be reviewed by my supervisor, and second because it is to be maintained by the Open Source community after its completion.

Phase 6: Demonstration phase
This phase was necessary because a demonstration a requisite for assessment.. In this phase, I created a utility application that was used for ad hoc validation of the results of the Gora operations directly from the Oracle NoSQL database. I also customised the LogManager application that Gora uses for demonstration purposes. Finally, I created a slide presentation that is also available online here: http://prezi.com/vnahscdp9vm4/apache-gora-support-for-oracle-nosql/

Phase 7: Documentation phase
Finally, after the presentation and demonstration, I wrote this detailed report that reviews and presents all the aspects of this project.

8.2 Project schedule

Following is a rough time schedule of the project, as the phases were actually performed.
June 1st – June 15th: Phase 1 - Initiation and planning phase
June 16th – June 26th: Phase 2 - Design phase
June 27th – Aug 27th: Phase 3 - Implementation phase
June 27th– Aug 27th: Phase 4 - System testing and bug fixing phase
Aug 28th – Aug 29th: Phase 5 – Code clean-up phase
Aug 30th – Sept 2nd: Phase 6 – Demonstration phase
June 1st– Sept 12th: Phase 7 – Documentation phase

Note that the Phase 4 and Phase 5 are concurrent phases. Also note that the Documentation phase run throughout the project's lifetime.

8.3 Risk Analysis

During the planning phase of this project we performed a risk analysis to identify potential high risk areas. This analysis also played a key role in performing the feasibility assessment of the project (see section 4.3). In the following sections, we will present the identified risks. Next we will assess each of them and provide the proposed solutions for each one.

8.3.1 Risk identification

The following list presents the identified risks. Each risk is numbered for future reference.

1. Zero previous knowledge of NoSQL technologies might require steep learning curve before become productive.
2. Apache Gora, as a very new Open Source project, might have several bugs that could delay successful integration.
3. Object to key/value store mapping might not be possible.
4. Apache Gora and Oracle NoSQL might have APIs that are incompatible with each other.
5. Apache Gora might lack documentation and prohibit us from learning its internal functionality.

8.3.2 Risk assessment and management

Following we will assess each identified risk and provide risk strategies for each one. To evaluate the risk, we established a scale from 1 to 10 for both the risk probability and the risk impact. 10 is considered to be the most probable or the one with the highest impact.

1. Probability of risk: 10. Impact: 10. Risk = 100.
   Risk strategy: Establish proper time management and prioritise the topics that should be studied. Set achievable small goals and proceed at a fast pace. Communicate frequently with the supervisor about the progress. Main focus should be the actual NoSQL technologies that will be used in this project. Use trusted resources such as books of renowned authors.
2. Probability of risk: 3. Impact: 6. Risk = 18.
   Risk strategy: Create mock implementations of the Gora interfaces in order to make sure that an integration is feasible. Study previous Gora datastores to identify bugs that delayed these projects and verify that they are resolved. If not, communicate with the Gora community to stress the need to resolve them.
3. Probability of risk: 1. Impact: 7. Risk = 7.
   Risk strategy: Study existing Gora datastores that have key/value data model (such as the Gora-DynamoDB module) and learn how they achieved the mapping. Also, research if other projects achieved object persistence in other key/value stores. The probability of risk is low because the existing Gora-DynamoDB datastore module provides similar mapping.
4. Probability of risk: 2. Impact: 9. Risk = 9.
   Risk strategy: Study the Gora API and the Oracle API. Create early prototypes to assess compatibility. Research other projects that have managed to integrate with these systems. In the worst case of inevitable incompatibility, use a model transformation framework such as SiTra.
5. Probability of risk: 8. Impact: 7. Risk = 56.
   Risk strategy: Gather all available documentation. Communicate with the Gora community and ask for resources. Study the source code of Gora. Solve known bugs and provide improvements of the existing functionality in order to learn the Gora architecture and its internal functionality. Create bonds with the other Gora developers.

Chapter 9

Conclusion

In this project we focused on exploring the NoSQL field. Our main objective was to prove that an object to a key/value store mapping is possible and that object persistence can be achieved in such a data store. We proved this concept by extending the datastore capabilities of the Apache Gora framework so it can be integrated with the Oracle NoSQL Database. To achieve this, we first studied thoroughly the various NoSQL technologies and data models. Then, we studied extensively the functionality, the architecture and the internals of the Apache Gora framework. Next, we studied the Oracle NoSQL Database, its data model and its API in order to properly use it as the back-end data storage mechanism for the Gora-OracleNoSQL datastore module, which was the main deliverable of this project.

The Gora-OracleNoSQL datastore module not only provides full datastore capabilities to the Apache Gora but also provides several extra features such as additional CRUD operations, ACID transactions and a Result Cache mechanism. This module allows Apache Gora to be used in enterprise infrastructures and integrate seamlessly with the Oracle technology stack, including the market leader Oracle RDBMS.

This self-proposed project makes me feel proud and honored for two reasons. First is the fact alone that my project proposal was accepted by Google's Summer of Code 2013. It has been a great pleasure to participate in this great event where I have gained valuable experience and knowledge. Apart from the technical knowledge that I have acquired, I have also improved other important skills such as self-discipline, self-organization and self-motivation. The second reason is that due to the overall significance of the project's contribution to the Apache Gora project, I have been nominated to be promoted to the Apache Project Management Committee. I believe that this project has in its entirety been a personal and professional success.

This project is a contribution to a very well-known Open Source Foundation: the Apache Software Foundation. It is my sincere hope that this system will be used by the Open Source community and the enterprise application developers. Personally, I will continue to maintain it in an attempt to help the Apache Gora become the standard persistence framework for Big Data.

Bibliography

[1] Pramod, J. Sadalage and Fowler, M. (2012). “NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence”. 1st edition. Addison-Wesley Professional

[2] Obasanjo, D. (2009). “Building scalable databases: Denormalization, the NoSQL movement and Digg”. Retrieved July 15th, 2013. Available at: http://www.25hoursaday.com/weblog/CommentView.aspx?guid=324e0852-ba72-4cc4-94bb-66b553fda165

[3] Storage Magazine, Storage Awards (2013). Retrieved August 19th, 2013. Available at: http://www.storagemagazine.co.uk/storageawards/index.php?page=winners201

[4] Fontecchio, M. (2013). “Oracle the clear leader in $24 billion RDBMS market”. TechTarget. Retrieved July 27th, 2013. Available at: http://itknowledgeexchange.techtarget.com/eye-on-oracle/oracle-the-clear-leader-in-24-billion-rdbms-market/

[5] Apache Gora Design Goals. Retrieved August 30th, 2013. Available at: ht-tps://github.com/enis/gora/wiki/Design

[6] Apache Gora tutorial. Retrieved June 4th, 2013. Available at: http://gora.apache.org/index.html

[7] Joshi, A. Haradhvala, S. Lamb, C. (2012). “Oracle NoSQL Database – Scalable, Transactional Key-value Store”. Oracle Corporation. Burlington. Retrieved July 9th, 2013. Available at: https://issues.apache.org/jira/secure/attachment/12583735/immm_2012_4_10_20050.pdf

[8] Oracle Corporation, (2013). “Getting Started with NoSQL Database”. Oracle Corporation World Headquarters. Retrieved August 6th, 2013. Available at: http://docs.oracle.com/cd/NOSQL/html/GettingStartedGuide/Oracle-NoSQLDB-GSG.pdf

[9] Sommerville, I. (2010). “Software Engineering”. 9th edition. Addison-Wesley

[10] Miller, J. (2009). "Design For Convention Over Configuration". Microsoft. Retrieved August 4th, 2013. Available at: http://msdn.microsoft.com/en-us/magazine/dd419655.aspx

[11] Koskela, L. (2007). “Test Driven: Practical TDD and Acceptance TDD for Java Developers”. Manning Publications Co., Greenwich, CT, USA

[12] Pirzadeh, P.; Tatemura, J. Hacigumus, H. (2011). "Performance Evaluation of Range Queries in Key Value Stores and Parallel and Distributed Processing Workshops and PhD Forum (IPDPSW)”. IEEE International Symposium, vol.1092, no.1101, pp.16-20

[13] Hilbert, M. “Big Data for Development: From Information- to Knowledge Societies” Retrieved July 17th, 2013. Available at: http://papers.ssrn.com/sol3/papers.cfm?abstract_id=2205145

[14] Oracle Corporation, (2012). “Oracle Information Architecture: An Architect’s Guide to Big Data.”. Retrieved July 20th, 2013. Available at: http://www.oracle.com/technetwork/topics/entarch/articles/oea-big-data-guide-1522052.pdf

[15] Laney, D. (2001). "3D Data Management: Controlling Data Volume, Velocity, and Variety". META Group. Retrieved June 3rd, 2013. Available at: http://blogs.gartner.com/doug-laney/files/2012/01/ad949-3D-Data-Management-Controlling-Data-Volume-Velocity-and-Variety.pdf

[16] Dorband, J. Palencia, J. and Ranawake, U. (2003). "Commodity computing clusters at Goddard Space Flight Center, Journal of Space Communication". pp: 113-123.

[17] Oracle Corporation, (2013). "Big Data for the Enterprise". Retrieved August 21st, 2013. Available at: http://www.oracle.com/us/products/database/big-data-for-enterprise-519135.pdf

[18] Vaish, G. (2013). "Getting Started with NoSQL". Packt Publishing.

[19] Segleau, D. "On Oracle NoSQL Database". Retrieved July 19th, 2013. Available at: http://www.odbms.org/blog/2013/07/on-oracle-nosql-database-interview-with-dave-segleau/

[20] Apache Avro 1.7.5 Documentation. Retrieved June 1st, 2013. Available at: http://avro.apache.org/docs/current/

[21] Cabibbo, L. (2013) "ONDM: an Object-NoSQL Datastore Mapper". Faculty of Engineering, Roma Tre University. Retrieved June 15th, 2013. Available at: http://cabibbo.dia.uniroma3.it/pub/ondm-demo-draft.pdf

[22] Rodgers, U. (1989). "Denormalisation - Why, What and How?, Database Programming and Design"

[23] Bahsoon, L. (2013) "Evaluating Software Products". University of Birmingham. Retrieved on September 2nd, 2013. Available at: http://www.cs.bham.ac.uk/∼rzb/Evaluation%20Bahsoon.pdf

Appendix A

Deliverables

Project Report:

- https://issues.apache.org/jira/browse/GORA-217

Source Code:

- https://github.com/maestros/gora-oraclenosql - This directory contains the whole project. Please note that this includes the whole Apache Gora framework along with the source code of the Gora-OracleNoSQL datastore module. It is included because it contains the required core Gora functionality.
- https://github.com/maestros/gora-oraclenosql/tree/master/gora-oracle - This is the directory of my source code for the **Gora-OracleNoSQL datastore module** (see chapter 5). The actual source code files that have been created by me, contain javadoc comments that state me as the author. The source code of this module is also available online: https://github.com/maestros/gora-oraclenosql/blob/master/gora-oracle/. More specifically, the source code files that I created are the following:
  - /src/main/java/org/apache/gora/oracle/store/OracleStore.java
  - /src/main/java/org/apache/gora/oracle/store/OracleMapping.java
  - /src/main/java/org/apache/gora/oracle/store/OracleStoreConstants.java
  - /src/main/java/org/apache/gora/oracle/query/OracleResult.java
  - /src/main/java/org/apache/gora/oracle/query/OracleQuery.java
  - /src/main/java/org/apache/gora/oracle/util/OracleUtil.java
  - /src/main/java/org/apache/gora/oracle/util/OracleTerminal.java
  - /src/test/java/org/apache/gora/oracle/store/TestOracleStore.java
  - /src/test/java/org/apache/gora/oracle/GoraOracleTestDriver.java

  I have also made significant modifications to the publicly available LogManager front-end application. It is located in: gora-oraclenosql/gora-tutorial/src/main/java/org/apache/gora/tutorial/log/LogManager.java

- https://github.com/maestros/gora-oraclenosql/tree/master/contribution-patches - This is the directory that contains the source code of the patches that I contributed to the Apache Gora framework (see Chapter 6). Inside the directory, except of the patch files, there is a readme file that provides some information for each patch. These patches have also been submitted to the Gora community and have been accepted and committed to the trunk. They are also available online: https://issues.apache.org/jira/browse/GORA.

Appendix B

How to run the software

Before anything else, the Gora framework should be installed. In order to install the Gora framework and to run the testing suite, the Apache Maven 2 software has to be installed in the system. If Maven is installed, then Gora has to be installed as follows:

```
gora−oraclenosql$ mvn clean install −Dmaven.test.skip=true
```

After the completion of this command a “SUCCESS” message should me displayed next to each Gora module. At the bottom, Maven should display “BUILD SUCCESS”.

The Apache Gora is a persistence framework and the Gora-OracleNoSQL is a datastore module of the Gora framework. As a framework, it cannot be “run” by itself. There are 2 ways of validating and/or demonstrating the functionality of the Gora-OracleNoSQL datastore module.

The first way, is to run the testing suite which executes all the unit tests. The unit tests perform several CRUD operations, using several setups, on several persistent objects of various data beans. After the execution of all the unit tests, the database is truncated by calling the OracleStore.deleteSchema() of each created DataStore. Therefore, the back-end database will be empty after the completion of the testing suite execution. Note that the testing suite has been successfully tested under both GNU/Linux and Mac OS X. Note that in order to run the testing suite, there is no need to manually start the OracleNoSQL server, because an embedded server is provided. Following are the commands to execute on a shell (terminal):

```
gora−oraclenosql$ cd gora−oracle
gora−oraclenosql/gora−oracle$ mvn clean test
```

The output of this command will be the logs generated by the unit tests. At the end the following should be displayed:

```
Results :
Tests run: 36, Failures: 0, Errors: 0, Skipped: 2
```

Note that the 2 skipped tests are the testGetPartitions() and the testDeleteByQueryFields() for the reasons that have been already discussed (see sections 5.5 and 7.4). In case the tests failed because of connectivity issues with the server, there might be an Operating System specific issue that prohibits the embedded server to be executed. In such case make sure that no server process runs and execute manually the server, as described later.

The second way of demonstrating the functionality of the Gora-OracleNoSQL datastore module, is by the use of the LogManager. As this is an external front-end application that uses the Apache Gora framework, the Oracle NoSQL server needs to be manually started. To start the server, execute the following commands in a separate shell session:

```
gora−oraclenosql/gora−oracle$ java −jar lib−ext/kv−2.0.39/kvstore.jar kvlite
```

Next, we execute the LogManager, from a different shell session, as follows:

```
gora-oraclenosql$ bin/gora logmanager
```

This will display the available options of the LogManager. Note the path from which the LogManager is executed. The LogManager is pre-configured to use the Gora-OracleNoSQL datastore module as the default Gora datastore. For a detailed description of the functionality of each option of the LogManager see the section 5.6. Note that we execute the LogManager from the *gora-oraclenosql* directory (the root directory of the project).

Following we present some sample commands that can be executed using the LogManager:

```
bin/gora logmanager -parse gora-tutorial/src/main/resources/access.log
bin/gora logmanager -get 9998
bin/gora logmanager -get 9998 ip
bin/gora logmanager -get 9998 ip,url
bin/gora logmanager -get 9998 ip,referrer,url
bin/gora logmanager -query 9996
bin/gora logmanager -delete 9998
bin/gora logmanager -query 9998
bin/gora logmanager -query 9995 9997
bin/gora logmanager -deleteByQuery 9995 9999
bin/gora logmanager -query 9995 9999
```

Note that the *-parse* option calls the OracleStore.put() method that persists the parsed objects.

To validate the results of the LogManager directly from the database, we can use the OracleTerminal utility application that provides a simple terminal that connects directly to the database server. For a detailed description of the functionality of each option of the OracleTerminal see the section 5.6. The OracleTerminal runs inside Maven, which properly sets up the environment variables. To use the OracleTerminal, execute it from the *gora-oraclenosql/gora-oracle* directory.

Next, we present some sample commands that can be executed using the OracleTerminal.

```
mvn exec:java -Dexec.mainClass="org.apache.gora.oracle.util.OracleTerminal"
-Dexec.classpathScope="compile" -Dexec.args=""

mvn exec:java -Dexec.mainClass="org.apache.gora.oracle.util.OracleTerminal"
-Dexec.classpathScope="compile" -Dexec.args="dumpDB"

mvn exec:java -Dexec.mainClass="org.apache.gora.oracle.util.OracleTerminal"
-Dexec.classpathScope="compile" -Dexec.args="get AccessLog/9998"
```

Note the "*AccessLog/9998*" parameter in the last command. The "AccessLog" is the "schema" of the DataStore that the LogManager uses. This is defined in the mapping file, which is located in: */gora-oraclenosql/gora-tutorial/conf/gora-oracle-mapping*. The "*9998*" is the key of the persistent object.

To delete all the key/value pairs from the database use the following command:

```
mvn exec:java -Dexec.mainClass="org.apache.gora.oracle.util.OracleTerminal"
-Dexec.classpathScope="compile" -Dexec.args="truncate"
```

After the execution of this command, the database will be empty.

Appendix C

Progress Reports

Following are the two progress reports that were submitted to my mentor during the Google Summer of Code 2013. These are also available online at the URL: http://svn.apache.org/repos/asf/gora /committers/reporting/gsoc2013/gora-oracle/.

Report # 1
Project Name: Apache Gora support for Oracle NoSQL datastore
Project URL: https://issues.apache.org/jira/browse/GORA-217
Report compiled by: Apostolos Giannakidis
Report date: July 5, 2013

Project Description
Further expanding Apache Gora's datastore support is key for becoming a standard persistence framework. The goal of this project is to extend the integration capabilities of Apache Gora. By implementing a new module for Gora, named Gora-Oracle, this project aims to offer a new NoSQL datastore which will enable Apache Gora users and developers to utilize the expressiveness offered from the Gora framework in conjunction with the functionality of the enterprise-class Oracle NoSQL database. Oracle NoSQL is as a distributed, highly scalable, low latency, key-value database for real-time big data workloads.

Checklist:
1. Objective: Study and review Oracle NoSQL data model. Oracle NoSQL data model is more sophisticated than simple key-value pairs however it keeps the data model simple. More specifically, each key is composed of a Major Key Path (major components) and a Minor Key Path (minor components). Both the major and the minor components are consisted of lists of Java Strings. The major component of each key is mandatory and requires at least one String (component). The concatenation of the Major Key Path and the Minor Key Path consist the full key. This data model allows flexible data modeling that, when done properly, provides fast, indexed lookup. Values are opaque data items (hence no specific data types) and can be either arbitrary byte arrays or the Oracle recommended Avro data format, which allows more complex data structures. Values can be left empty.
2. Objective: Study and review the Oracle NoSQL API. Oracle NoSQL maintains detailed documentation and manuals that helped me familiarize myself with the product. CRUD operations are documented clearly along with code snippets that demonstrate the use of the API. The API provides several methods that perform CRUD operations. These are the following: put, putIfAbsent, putIfPresent, delete, deleteIfVersion. Also, there are many overloaded versions of each of them.
3. Objective: Deploy and run sample applications in Oracle NoSQL. This was done with ease. The deployment of the Oracle NoSQL merely involves downloading from Oracle's web site and untarring the file archive. I had some difficulties on my machine because I use Mac OS X and for some reason I could not execute properly the kvstore server. I had to create an Ubuntu Virtual Machine, install Java 1.7 and properly configure the 2 virtual network adapters (Host-only adapter and NAT) in order to have access from and to the host machine and also to the Internet. After that

I could properly execute the kvstore server and successfully run the example applications provided by Oracle.

4. Objective: Identify dependencies and how they should be handled in this project. This module depends on the kvclient.jar of the Oracle NoSQL database. This jar is the driver that provides the necessary Oracle NoSQL API that Apache Gora will use. We identified that there is a license incompatibility issue that prohibits us to incorporate this jar file in the distribution of Apache Gora. I am in contact with the Oracle NoSQL development and management team and they informed us that they are working on a possible solution. In case no solution comes from Oracle, the module will have to be downloaded separately of the main Gora distribution and the user will be responsible to download the Oracle NoSQL database and install the kvclient.jar in the proper directory of Gora's file system.

5. Objective: Investigate existing solutions that integrate with Oracle NoSQL. Apart from the official integration of Oracle NoSQL with Apache Hadoop, Oracle Coherence, and the Oracle RDBMS (with the use of External Tables), I have also identified a few of other projects that integrate with the Oracle NoSQL. The most important of them are:

a) Blueprints – A property Graph Model Interface The blueprints-oracle-nosqldb is a Java implementation of the Blueprints API over Oracle NoSQL database, created by Daniel McClary. Project URL: https://github.com/dwmclary/blueprints-oracle-nosqldb

b) Hive - A data warehouse system for Hadoop The HiveKVStorageHandler2 is a Java implementation of a Storage Handler to query data stored in Oracle NoSQL Database via Hive, created by Alexandre Vilcek. Project URL: https://github.com/vilcek/HiveKVStorageHandler2

c) Zorba – A NoSQL Query Processor The oracle-nosql-db-module is a C++ external module for using the Oracle NoSQL Database within Zorba, created by the Zorba Coders team. Project URL: https://launchpad.net/zorba/oracle-nosql-db-module

6. Objective: Create file structure for the Gora-Oracle module. My mentor and me decided that this should be exactly the same as the other modules. We also agreed to rename the module from Gora-OracleNoSQL to Gora-Oracle. All the necessary changes to the file structure and the pom files have already been made.

7. Objective: Start coding some of the classes. I have created a Github repository for this project (github.com/maestros/gora-oraclenosql) and I have properly set up SVN and remote Git repositories in order to be able to fetch the trunk changes. I have created the necessary java files and I have started coding the OracleStore, TestOracleStore and GoraOracleTestDriver. However, a major drawback emerged that does not allow us to continue working on the GoraOracleTestDriver. The drawback is the fact that Oracle NoSQL does not provide an embedded kvstore server that is needed. My mentor and me are looking for alternative ways to tackle this issue.

8. Objective: Contribute to the community by addressing Jira issues. I have already worked in the following Jira issues:

• GORA-243: Properly escaping spaces of GORA_HOME in bin/gora I identified a bug, which I documented it in a new Jira issue and provided a patch that solves the issue. The patch has already been accepted.

• GORA-255: Remove deprecated methods from DataStoreTestBase I identified an improvement that should be performed in the DataStoreTestBase and documented it in a new Jira issue. I assigned the issue to me but I did not have the time yet to work on it. I am also waiting for some feedback from the community before I start working on this.

• GORA-229: Use @Ignore for unimplemented functionality to identify absent tests Issue reported by Lewis McGibbney that I assigned it to myself. It involves refactoring all the Java classes of Apache Gora to abandon the deprecated junit.framework.* and use the recommended static org.junit.Assert.* of JUnit 4.10. It also involves adding the @Ignore annotation to all the test methods that do not have proper implementation yet. I performed all the necessary work and provided 4 patches - one for each Gora module. I am waiting for feedback on these patches from the Gora community.

9. Objective: Create a social network around this project. I am actively communicating about this project in Gora's Jira, Oracle Forums, Linkedin groups and contacts and also via my personal blog (www.giannakidis.info). It is evident that I already have a backlink from an Oracle's employee blog to one of my blogposts.

Next Steps:

1. We have to decide how can all the CRUD method variation be mapped to Apache Gora.

2. Find a proper way of addressing the lack of the embedded kvstore server in order to continue with the implementation of the GoraOracleTestDriver. Currently we are considering of using Atlassian's bash-maven-plugin that will allow executing bash scripts using maven. This way we could execute the kvstore server within maven. Because this plugin is not available from Maven Central we have to work on pushing it there.

3. Finish working with the GoraOracleTestDriver and continue to the TestOracleStore and the OracleStore.

4. Decide if we need to upgrade Gora's Avro dependency because Oracle NoSQL uses a more recent version of Avro, which might conflict with Gora's one, which uses an older one.

5. Study the mappings that the other Gora modules use and get a better understanding of Oracle NoSQL data model and modeling capabilities in order to create a proper mapping for this module.

6. Continue working on existing Jira issues and create new ones were appropriate. More specifically, I will work at the following Jira issues:

   a. GORA-241: Properly document WebServiceBackedDataStore Interface

   b. GORA-232: DataStoreTestBase should delegate all functionality to DataStoreTestUtil

   c. GORA-255: Remove deprecated methods from DataStoreTestBase.

7. Continue working on establishing a social network around this project by publishing more blog posts and further communication in other social channels.

Supervisor's Comments

Apostolos has shown a high degree of interest and patience with the project to date. His understanding of Gora is coming along very nicely and he is showing competence in his use of version control e.g. Subversion and Git to skip between code bases as required. With regards to his approach to the project, he has been in very regular contact with me and we have been discussing aspects of the project regularly. This seems to be working well. As the report and it's content displays, Apostolos was a great fit for this project and I am looking forward to working on the next steps as described above. We will progress on this basis for now.

Supervisor: lewismc

Report # 2
Project Name: Apache Gora support for Oracle NoSQL datastore
Project URL: https://issues.apache.org/jira/browse/GORA-217
Report compiled by: Apostolos Giannakidis
Report date: 1 August, 2013

Project Description
Further expanding Apache Gora's datastore support is key for becoming a standard persistence framework. The goal of this project is to extend the integration capabilities of Apache Gora. By implementing a new module for Gora, named Gora-Oracle, this project aims to offer a new NoSQL datastore which will enable Apache Gora users and developers to utilize the expressiveness offered from the Gora framework in conjunction with the functionality of the enterprise-class Oracle NoSQL database. Oracle NoSQL is as a distributed, highly scalable, low latency, key-value database for real-time big data workloads.

Checklist:
1. Objective: Study and review the Oracle NoSQL API. Apart from the methods that provide CRUD operations for a single key, Oracle NoSQL provides an efficient and transactional mechanism for executing a sequence of operations associated with keys that share the same Major Path. This feature allowed me to implement the datastore in such a way that it batches the operations before executing them. Then, when the flush() method of the datastore is called, all the batched operations are executed in Oracle NoSQL.

2. Objective: Deploy and run sample applications in Oracle NoSQL. At first I had some issues running the Oracle NoSQL server on my host OS (Mac OS X) because Oracle does not support, officially, this OS. Therefore, initially, I solved this problem by installing the Oracle NoSQL server in a Virtual Machine. However, this solution could not work efficiently as I had major difficulties running the Gora test cases from my host OS against the Oracle NoSQL server, because Gora uses an embedded Test Driver, which, at first, could not be executed on my host OS. Thus, I investigated the issue further and I managed to successfully deploy and execute Oracle NoSQL on Mac OS X. I blogged about this in my personal blog: www.giannakidis.info/post/54750399281/running-oracle-nosql-on-mac-os-x

3. Objective: Design of the module. My mentor and me agreed that I should focus more on the coding part rather than on the UML modeling. Also, I designed a mapping scheme on how to persist a POJO in Oracle NoSQL, which uses a simple key/value pair data model. The mapping scheme involves the elaborated nature of the Oracle NoSQL key, which is composed of one or more major components and one or more minor components. This way, I used the first major component to be treated as a table, the second major component to be treated as a record primary key, the first minor component to be treated as a column family, if any, and the second minor component to be treated as a field. It seems to be a straightforward mapping scheme, which has also been approved by my mentor.

4. Objective: Find a solution to the embedded Oracle NoSQL server issue. One of the pending tasks since the previous report, was to find a proper way of addressing the lack of the embedded kvstore server in order to continue with the implementation of the GoraOracleTestDriver. At first, we were considering of using Atlassian's bash-maven-plugin that will allow executing bash scripts using maven. This way we could execute the kvstore server within maven. However, because this plugin is not available from Maven Central, I was also working on an alternative solution. This solution involves the manual creates (spawn) of a new process that executes the Oracle NoSQL server (kvstore.jar) and its manual termination. This solution seems to work successfully in my local machine.

5. Objective: Finish working on the GoraOracleTestDriver Because of the working solution of the embedded Oracle NoSQL server, I managed to complete the implementation of the GoraOracleTestDriver. Some minor enchantments are pending such as the deletion of the temporary files after it finishes its execution.

6. Objective: Implementation. I have moved on with the implementation of the mapping, as described above. The mapping is almost complete. Some minor improvements might be needed to be done in the next weeks. Also, working under a Test-Driven methodology, I have been working towards an implementation of the put() and get() methods that will allow the CURD unit tests to

pass successfully. Until now, I have completed the implementation of put() and get() in such a way that assertPutBytes() completes successfully. To facilitate the implementation of these methods (and the creation of the unit tests) I have created a helper class (OracleUtil) that contains static utility methods. Finally, based on some feedback I had from my mentor, I made some adjustments to the source code such as adhering to the indentation of the Gora codebase and including javadocs for every method.

7. Objective: Contribute to the community by addressing Jira issues. I have already worked in the following Jira issues:

• GORA-229: Use @Ignore for unimplemented functionality to identify absent tests Issue reported by Lewis McGibbney that I assigned it to myself. It involves refactoring all the Java classes of Apache Gora to abandon the deprecated junit.framework.* and use the recommended static org.junit.Assert.* of JUnit 4.10. It also involves adding the @Ignore annotation to all the test methods that do not have proper implementation yet. I performed all the necessary work and provided 4 patches - one for each Gora module. The patch has already been accepted and committed.

• GORA-232: DataStoreTestBase should delegate all functionality to DataStoreTestUtil Issue reported by Lewis McGibbney and I assigned it to myself. It involves refactoring the unit test methods that contain test code of the DataStoreTestBase and delegate their functionality to the utility class DataStoreTestUtil. I worked on this issue and provided a patch that delegates the functionality of the following test methods: testPutNested(), testPutArray(), testPutBytes(), and testPutMap(). The patch has not yet been committed.

• GORA-258 writeCapacUnits gets value from wrong attribute Bug identified by me. I documented it in a new Jira issue and provided a patch that solves the issue. The issue involves a variable that reads a value from a wrong property in the Gora-DynamoDB module. The patch has already been accepted and committed.

• GORA-259 Removal of the main methods from the test case classes Issue identified by me. I documented it in Jira and I assigned it to myself. I identified that several JUnit test case classes contain main methods, which is a non-recommended practice. After I received positive feedback from the community, I proceeded to create the patch, which has already been committed.

• GORA-264 Make generated data beans more java doc friendly Issue identified by Renato Javier Marroquín Mogrovejo. I assigned it to myself and I provided a patch for the GoraCompiler that adds javadocs to the generated data beans. Some of the javadoc that is added is static and some is dynamic, taken from the json avro schema. Therefore, I extended the json avro schema to also understand and use the "doc" attribute. The patch has not yet been committed.

8. Objective: Create a social network around this project. I continue to actively communicate about this project in Gora's Jira, Oracle Forums, Linkedin groups and contacts and also via my personal blog (www.giannakidis.info). This project has attracted some attention and some publicity. The most important example is a wonderful blog-post of Steve O'Hearn, author of the book "OCA Oracle Database SQL Expert Exam Guide: Exam 1Z0-047" (Oracle Press), which talks with very warm words about this project. http://blog.corbinian.com/node/89

Next Steps:

1. There are some CRUD methods in Oracle NoSQL that provide flexibility regarding consistency, durability and several storage and query options. We have to decide how and if these methods can be mapped to Apache Gora.

2. Remove, on clean up, the temporary files created by the GoraOracleTestDriver when it spawns a new process for the Oracle NoSQL service.

3. Continue working on the TestOracleStore and the OracleStore and provide implementations for the delete and Query methods and classes. Ideally, these implementations should be finished in the next month.

4. Decide if we need to upgrade Gora's Avro dependency because Oracle NoSQL uses a more recent version of Avro, which might conflict with Gora's one, which uses an older one.

5. Continue the studying of the implementations of other Gora modules and get a better understanding of how the CRUD operations are implemented in other datastores.

6. Continue working on existing Jira issues and create new ones were appropriate. I will work at the following Jira issues, for which I would like some community feedback before I proceed to work on the patches:

a. GORA-241: Properly document WebServiceBackedDataStore Interface
b. GORA-255: Remove deprecated methods from DataStoreTestBase

7. Continue working on establishing a social network around this project by publishing more blog posts and further communication in other social channels.

Supervisors Comments

Apostolos' project has come on leaps and bounds since last reporting. There is clear evidence that he has a firm grasp on Gora as a community project and of course his GsoC project as a personal project. Apostolos has been very active within community communication as well as active on other social media streams to promote his project and to ask for help if and where it is required.

What he has achieved (since last reporting) is accurately documented above. What he hopes to achieve (next steps as above) is in agreement with what we have discussed in private.

His participation in Gora community issues is extremely encouraging and it is really nice to see a fresh injection of passion in to Gora.

I am looking forward to seeing the codebase progress and for the number of @Ignore test cases reduce over the second term.

Supervisor: lewismc