Politecnico di Milano

Computer Science and Engineering

Scuola di Ingegneria Industriale e dell'Informazione

Computer Science and Engineering

Avoiding CRUD operations lock-in in NoSQL databases: extension of the CPIM library

Advisor: Elisabetta DI NITTO

Co-Advisor: Marco SCAVUZZO

Master thesis by:

Fabio ARCIDIACONO matr. 799001

Academic Year 2013-2014

dedica. . .

Ringraziamenti

Ringraziamenti vari, massimo una o due pagine.

Milano, 29 Aprile 2015

Fabio.

Abstract

Within the last years, especially for web applications, data requirements have changed drastically; applications needs to handle information that are not always well structured and, more importantly, their volume is not sustainable for traditional data management techniques. Solutions that tries to handle those new kind of data have emerged over the classical DBMS solutions; those solutions comes under the name of NoSQL (Not Only SQL), to underline the different approach they bring with respect to traditional DBMS.
Many NoSQL databases has been developed in these years and, each of them, uses a different approach to handle the previously mentioned requirements. Those technologies provides a set of proprietary API that move toward the user a lot of programming effort with respect to DBMS solutions.
The lack of a common language for NoSQL databases, require a clear understanding of the available NoSQL solutions, to be able to choose the right technology for the application requirements. However, during the life cycle of the application, changing the adopted NoSQL technology, maybe due to changes in requirements or in business, may become a problem. This problem is known as *vendor lock-in*.
This work proposes a model that, using the CPIM library and through the JPA interface, permits to the users to develop applications using a common interface to interact with many NoSQL technologies and thus achieve code probability, leveraging the complexity of NoSQL systems while exploiting the advantages that those technologies bring in terms of scalability and performance. Moreover this work proposes the integration, in the CPIM library, of *Hegira* a migration and synchronization system to handle data migration and synchronization among NoSQL database.

Estratto

Negli ultimi anni, specialmente per le applicazioni web, i requisiti sulla gestione dei dati sono drasticamente cambiati; le applicazioni devono gestire dati che per loro natura non sono strutturati e, principalmente, vengono generati in una quantità tale che i sistemi tradizionali per la gestione dei dati non sono pi sufficientemente performanti. Varie soluzioni sono state proposte come alternative ai classici DBMS; queste soluzioni prendono il nome di NoSQL (Not Only SQL), per sottolineare il differente approccio adottato rispetto ai tradizionali DBMS.
Molti database NoSQL sono stati sviluppati in questi anni e, ognuno di essi, utilizza un approccio differente nel cercare di soddisfare i requisiti sopra citati. Queste tecnologie forniscono un insisme di API che mettono l'utente in condizioni di dover scrivere molto più codice rispetto a quanto sarebbe necessario utilizzando i tradizionali DBMS. La mancanza di un linguaggio comune a tutti i database NoSQL richiede una chiara e precisa conoscenza delle soluzioni disponibili sul mercato, per essere in grado di scegliere la tecnologia che più soddisfa i requisiti dell'applicazione. Tuttavia, durante il ciclo di vita di un'applicazione, cambiare la soluzione NoSQL adottata, ad esempio a fronte di un cambiamento nei requisiti o delle logiche di business, può essere un problema. Questo problema è noto come *vendor lock-in*.
Questo lavoro propone un modello che, usando la libreria CPIM e sfruttando l'interfaccia JPA, permetta all'utente di sviluppare applicazioni usando Jun'interfaccia comune per molte tecnologie NoSQL e ottenere così una buona protabilità del codice, mitigando la complessità delle tecnologie NoSQL senza perderne i vantaggi in termini di salabilità e performance. Inoltre questo lavoro propone l'integrazione, all'interno della libreria CPIM, del sistema di migrazione e sincronizzazione *Hegira*, per gestire la migrazione e la sincronizzazione dei dati fra database NoSQL.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

In the last few years, due to the advent of Web 2.0, more and more data become available, generated by a growing multitude of people. The nature of this kind of data is intrinsically unstructured and comes in a volume that traditional data management techniques are no more affordable to guarantee modern application requirements. In this scenario, NoSQL databases have emerged over traditional DBMS as a more suitable alternative to handle those new kinds of data.

NoSQL databases tries to address the new applications requirements in terms of: fault tolerance, availability across distributed data sources, scalability and consistency, in different ways, proposing different properties and characteristics. Each NoSQL database thus provides to its users a different API interface tailored to exploit the specific characteristic that the NoSQL offer.

The lack of a common language for NoSQL databases, require a clear understanding of the available NoSQL solutions, to be able to choose the right technology for the application requirements. However, during the life cycle of the application, changing the adopted NoSQL technology, maybe due to changes in requirements or in business, may become a problem. This problem is known as *vendor lock-in*.

To mitigate the problem, this work proposes an extension of the CPIM library, which already try to address the vendor lock-in problem in PaaS environments, in order to provide to the user a way to abstract from the specific NoSQL technology used to store the data. Many solutions has been proposed by both communities and industry, in defining a common way to access different NoSQL technologies. We propose to use the one that seems to get most in-

terest, especially by the industry, which is the use of the JPA interface. With this objective we integrate Kundera, an ORM for NoSQL databases based on the JPA standard interface, in the NoSQL service of the CPIM library. Furthermore we develop new extensions for Kundera to support the database currently supported by CPIM.
To achieve complete portability, both of the code and of the stored data, this work also propose the integration, in the CPIM library, of the migration and synchronization system *Hegira*. This gives to the user a way of moving its data from a NoSQL technology to another without experiencing any application down time, and thus be able to change the NoSQL technology without the need of re-engineering the application.

Original Contributions

This work include the following original contributions:

- two brand new Kundera clients, one for Google Datastore and one for Azure Tables;
- the support, for the NoSQL service of CPIM library, for the interaction with the migration system *Hegira*, both for data synchronization and migration;

Outline of the Thesis

This thesis is organized as follows:

- In Chapter 2 is described the evolution of NoSQL. As a first introduction is discussed why in this years this technology have emerged over SQL solutions and what are the main differences among those technology, the second part aims to underline the lack of a common language for NoSQLs in contrast to DBMS.
- In Chapter 3 we analyze the current implementation of the NoSQL service in the CPIM library; underlying the problem of the implementation and proposing a solution for each of them. Furthermore we explain why

we decided to integrate the migration and synchronization system *Hegira*, given the migration requirements of modern applications.

- Chapter 4 is dedicated to present the develop of the two Kundera client extension, developed to support Google Datastore and Azure Tables, in order to maintains the support of those databases in the CPIM library;

- In Chapter 5 is presented the work made on the CPIM library. As a first step are described the modifications in the CPIM NoSQL service aimed to integrate Kundera as unique persistence layer for NoSQL access using the standard JPA interface. Furthermore is discussed the extension of the CPIM library to include the required logic to interact with the migration and synchronization system *Hegira*.

- In Chapter 6 are described the various tests that have been performed on the developed Kundera extensions, to guarantee the correctness of the operation and to provide a measurement of performance in terms of latency and throughput. Finally is presented the application developed to test the interaction if the CPIM library with Hegira and the developed Kundera extensions.

- In Chapter 7 draws the conclusions on the entire work and proposes some possible future works.

Chapter 2

State of the art

2.1 Introduction

In this chapter NoSQL databases are firstly introduced, described in their main characteristics and a general classification is provided. In section 2.3 are listed some of the solutions that have been developed trying to define a common interface to interact with different NoSQL databases.
In section 2.3.3 is also described the JPA interface and JPQL, the querying language used for issuing queries. In section 2.4 the CPIM library is introduced as a tentative of defining a common interface to interact with different vendors in PaaS environments (which includes accessing their NoSQL solution).

2.2 NoSQL databases

NoSQL databases have recently became popular, due to the inadequacy of traditional RBDMSs to guarantee adequate performances with respect to the new application requirements that in these years have outlined. These requirements primarily concern the ability of handling huge amount of unstructured or semi-structured data.
RDBMSs provide a general-purpose solutions that balance the various requirements of applications, but they works well for application handling structured data and for applications that have strict requirement on consistency (as they guarantee full ACID consistency). Examples are business or administrative

application since RBDMSs were born more than 30 years ago, when these were the most common applications.
NoSQL databases have emerged as a way to store unstructured, semi-structured data and other complex objects such as documents, data streams, and graphs. Moreover they provide the capabilities for handling huge amount of those kind of data, while providing reasonable performance. This is done primarily by recognizing that many modern applications such as social networks, does not need full ACID compliance. Indeed, full ACID compliance may not be important to a search engine that may return different results to two users simultaneously, or to Amazon when returning sets of different reviews to two users.

2.2.1 NoSQL characteristics

Due to the different approaches that NoSQL database put in place to meet the new application requirements, there is no a unique NoSQL definition but since all of them provides similar features, we can say that NoSQL databases are distributed databases with flexible data models aimed to provide: horizontal scalability, fault tolerance and high availability.
As distributed systems, NoSQL databases are governed by the CAP theorem; it states that, in a distributed system, we can only have two out of three of the following guarantees:

- **Consistency**, a read is guaranteed to return the most recent write for a given client;
- **Availability**, a non-failing node will return a reasonable response within a reasonable amount of time (no error or timeout);
- **Partition tolerance**, the system will continue to function when network partitions occur.

As a result, we can have a distributed system with only one of these three combination of properties: CA, CP or AP.
Given that networks aren't completely reliable, partitions must be tolerated; according to the CAP theorem, this means that essentially two are the remaining valid options: Consistency and Availability.

For those reasons, while RDBMS scale very well vertically, and thus adding computational power to the node, they do not scale particularly well horizontally and thus through sharding. This is because RDBMS guarantee ACID properties, and thus consistency, while for sharding, the system should be tolerant to partition but, being a distributed system, this means, in terms of the CAP theorem, that RDMBS cannot guarantee availability. In contrast, NoSQL database sacrifice consistency for the sake of efficiency an thus, with respect to the CAP theorem, they can guarantee both partition tolerance and availability, properties that permit them to scale well horizontally by means of sharding.
The decision of neglecting consistency, makes NoSQL databases no more ACID compliant, to highlight this fact, they are said to be BASE compliant, where BASE is the acronym of:

- **Basically Available**, which indicates that the system does guarantee availability, in terms of the CAP theorem;
- **Soft state**, which indicates that the state of the system may change over time, even without input;
- **Eventual consistency**, which indicates that the system will become consistent over time, given that the system doesn't receive input during that time.

Those properties are perfect for certain type of web application, as pointed out previously, and permits to NoSQL databases to guarantee valuable performance in the scenario of Web 2.0 applications requirements.

2.2.2 NoSQL classification

Many NoSQL datastore has been developed and each of them manage data in different ways depending on the requirements it tries to address and how it ranks as compared to the CAP theorem. There is not a general classification for NoSQL databases, many are the method that can be used to classify them: the data model, the architecture, the sharding or replication methods. We will consider the classification based on the data model since is the most common method used to categorize them.

Accordingly to the meta model, NoSQL databases can be classified in four groups: key-value, column-oriented, document-oriented and grap-based.

Key/Value stores This kind of databases are very similar to data structures such as Maps or Hashtables, indeed, data are stored as values which are then associated to a key. This databases are thus completely schema-free and can easily store non structured data.This permits to write huge amounts of data and even to horizontally distribute them (sharded) among the nodes of the system. Examples of this kind of databases are Redis[1] and Voldemort[2].

Column-oriented databases Those databases are mainly inspired by the Google Big Table [13] data model, and are designed for storing data tables as sections of columns of data, rather than as rows of data. Hence, entity properties are stored in *Columns* that are then grouped in *Column families*; rows are identified by a key and composed by a set of column, each row can have a different set of columns with respect to other rows to be able to persist even semi-structured and unstructured data. This kind of databases allows great scalability options as they can both scale horizontally (sharding), by distributing rows, and vertically, by distributing column families among the node of the system.
Examples of this kind of databases are Cassandra[3] and HBase[4].

Document-oriented databases Document-oriented databases are designed for storing, retrieving, and managing document-oriented data. These systems are designed around an abstract notion of a Document identified in the database via a unique key. Document database typically offer a query language that allows the user to retrieve documents based on their content because they extract and index all kinds of meta-data and usually also the entire content of the documents. Documents can be stored in many different ways such as JSON, BSON, YAML or XML.
An example of this kind of database is MongoDB[5] and Couchbase[6].

[1] `http://redis.io`
[2] `http://www.project-voldemort.com`
[3] `http://cassandra.apache.org`
[4] `http://hbase.apache.org`
[5] `http://www.mongodb.org`
[6] `http://www.couchbase.com`

Graph databases Graph databases are databases that uses graph theory to represent the data. Persisted entities are represented by nodes; each node maintains the information about the entity it represents by storing them into properties, while edges can be used to represent relationships among the entities.
An example of this kind of database is Neo4j[7].

As we write there are more than 150 [3] different NoSQL databases and not all of them can be strictly categorized in one of the previous classification. Furthermore other solutions trying to propose multiple data models are being developed. Example of those systems are: OrientDB[8], an hybrid solution which data model span across graph and document databases and ArangoDB[9], which data model is an hybrid of document, graph and key-value data models.

2.3 Approaches for offering a common language over NoSQL

The variety of NoSQL systems is huge and the lack of a a common standardized language for NoSQL databases is a great concern for companies interested in adopting any of these systems, applications and data are expensive to convert and competencies and expertise acquired on a specific system get wasted in case of migration. Also, most of the NoSQL interfaces support a lower level of interaction than SQL, which appear to be a step back with respect to DBMS.

2.3.1 SQLifying NoSQL

A fist approach that is emerging is the *SQLfication* of NoSQL databases. NoSQL vendors, in order to overcome the problem of industry wasting the expertise maturated over SQL systems, started to create, around their NoSQL solutions, SQL-like wrappers, which typically offer different features with respect to those of a traditional relational database query language, but maintaining a grammar similar to that of SQL.

[7] `http://neo4j.com`
[8] `http://www.orientechnologies.com`
[9] `https://www.arangodb.com/`

For example Google App Engine Datastore, provides GQL, a SQL-like language for retrieving entities or keys from Datastore. Other NoSQL database, such as Cassandra (with CQL), or OrientDB, provide such type of SQL-like language support natively.

There exist also some independent projects that aim to create such kind of SQL-like languages upon existing NoSQL databases.

Apache Phoenix

Apache Phoenix [11] aims to become the standard means of accessing HBase data through a well-defined, industry standard API. It is a relational database layer over HBase delivered as a client-embedded JDBC driver over HBase data. Apache Phoenix takes standard SQL queries, compiles them into a series of HBase scans, and orchestrates the running of those scans to produce regular JDBC result sets.

UnQL

Unstructured Data Query Language [6], or UnQL (pronounced Uncle), is a tentative to bring a familiar and standardized data definition and manipulation language to the NoSQL domain. The project was started in 2011 by the joint effort of Couchbase and SQLite.
After the project was started, and after some burst of activity, the project came to a hold. So it seems, that at least as a project UnQL has been a failure.

2.3.2 Meta-model approaches

Another way that has been investigated in order to achieve interoparability, is trying to abstract from the data model by spotting common concepts in the data model of various NoSQL solutions, in order to define a more general meta-model that can offer an common interface to store data.

Apache MetaModel

The aim of Apache MetaModel is to provide a common interface for discovery, exploration of metadata and querying of different types of data sources [9].
The peculiarity of this project is that it does not only provide support for NoSQL database (such as CouchDB, MongoDB, Hbase, Cassandra and ElasticSearch) but also for relational databases (such as PostgreSQL, MySQL, Oracle DB and SQL Server) and even for various raw data format (such as JSON, XML and CSV files).
The defined meta-model gives the ability to add data sources at run-time and provides a type-safe SQL-like API. An example for a table creation is reported in the code 2.1, the meta-model defines a different semantic for the concept of *table* depending on the underlying storage technology. For example, for JDBC it issues a CREATE TABLE statement to the database, for a CSV file, it creates or overwrites the file and for MongoDB, it creates a new collection in the MongoDB database.

```
1 // CREATE a table
2 Table table = callback.createTable("Employee")
3     .withColumn("id").ofType(INTEGER)
4     .withColumn("name").ofType(VARCHAR).execute();
```

Listing 2.1: Apache MetaModel example

SOS Platform

The SOS (Save Our Systems) Platform [16] is an academic project developed by Unversità Roma Tre.
The platform achieves interoperability among NoSQL databases by defining a common interface through a meta-layer of abstraction used to maintain entities information. Database specific handlers read the meta-layer and translate the meta-operations to database-specific operations that are finally performed over the database instance. The architecture is shown in figure 2.1.
The supported NoSQL databases are: Hbase to represent Column-based databases, MongoDB to represent Document-oriented databases and Redis for Key/Value stores. The meta-layer exploit the data model similarities and is actually very simple; it is composed by three main constructs: Struct, Set and Attribute. Attributes contain simple values, such as Strings or Integers;

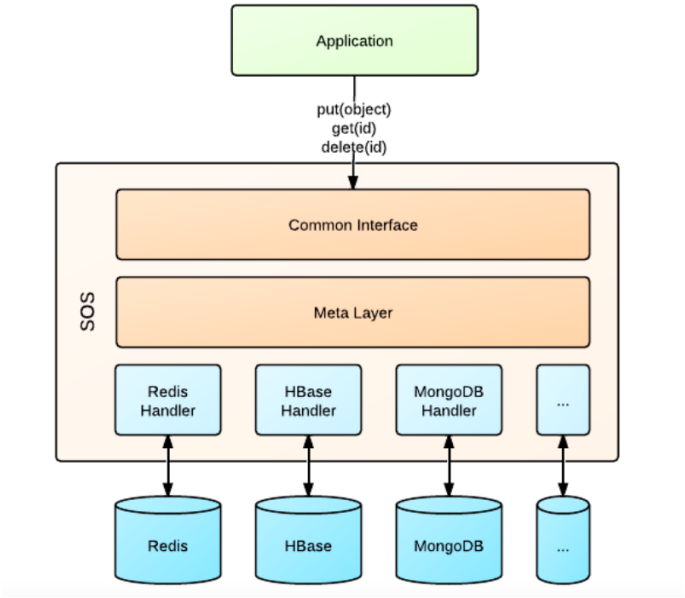


Figure 2.1: SOS architecture [16]

Structs and Sets are instead complex elements whose values may contains both Attributes and Sets or Structs as well. Each database is represented as a Set of collections whose is a Set itself, containing an arbitrary number of objects. Each object is identified by a key that is unique in the collections it belongs to.

2.3.3 ORM approaches

Object Relational Mapping (ORM) solutions came into existence to solve the *object-relational impedance mismatch* problem, that is often encountered when a relational database management system (RDBMS) is being used by a program written in an object-oriented programming language or style; ORM were thus primarily introduced to ease the interaction with RDBMS.
Each ORM solution had its own API and object query language (like HQL for Hibernate) which made it difficult for programmers to switch from one framework to another. As a result, efforts were made to make standards and specifications for this tools.
Today ORM have become the standard way that developers use to interacts with RDBMS due to the great advantages they bring, such as the ability to change RDBMS without modifications in the application code. The advantages

that ORM led to RDBMS made people think that this approach, or a similar one, should be applied to NoSQLs too. Furthermore people lack in-depth knowledge of NoSQL and even if they do, their knowledge is limited to a couple of them. For this reasons the ORM approach seems to fit perfectly since they gives to developers a way to interact with NoSQL in a way they are comfortable with, leaving the complexity of NoSQLs to the ORM.

The JPA interface

Since many of the ORM solutions have their roots in the JPA interface, an ORM standard for Java based applications, is worth describe it.

The Java Persistence API [15] was first released as part of Enterprise JavaBeans 3.0 in 2006. As a more general-purpose object-relational mapping facility, it was quickly recognized as such, and was expanded at the request of the community to support use in Java SE environments as well as in the other Java EE container types.
The Java Persistence API provides an object/relational mapping facility to Java developers for managing mainly relational data within Java applications. Java Persistence consists of three areas:

- the Java Persistence API;
- object/relational mapping meta-data;
- the query language.

Mapping meta-data are defined by the user as Java annotations upon the classes that he wants to map to the underlying database. The user annotates classes representing entities; typically, an entity represents a table in a relational database, and each entity instance corresponds to a row in that table. Entities are managed by the entity manager; the `EntityManager` API creates and removes persistent entity instances, finds entities by their primary key, and allows queries to be run on entities. The `EntityManager.createQuery` and `EntityManager.createNamedQuery` methods are used to query the database using Java Persistence query language queries; the only difference among them is that the latter permits to define queries statically, within the entity meta-data, through a specific annotation and referencing it later by name.

A *peristence unit* is set to all entity classes managed by the `EntityManager` instance. Persistence units are defined by the *persistence.xml* configuration file. Each persistence unit is identified by a name, that is unique across the persistence units scope.

The JPA supports two methods for expressing queries, in order to retrieve entities and other persistent data from the database: query languages and the criteria API. The primary query language is Java Persistence Query Language (JPQL), a database-independent query language that operates on the logical entity model, as opposed to the physical data model. Queries may also be expressed in SQL to take advantage of the underlying database. The criteria API provides an alternative method for constructing queries based on Java objects instead of query strings. An example of defining the same query with the two approaches is shown in the code 2.2. the JPA approach.

```
1 // using criteria API
2 CriteriaBuilder cb = em.getCriteriaBuilder();
3 CriteriaQuery<Employee> query = cb.createQuery(Employee.class);
4 Root<Employee> e = query.from(Employee.class);
5 query.select(e);
6
7 // using JPQL
8 TypedQuery<Employee> query = em.createQuery("SELECT e FROM Employee e");
```

Listing 2.2: Create queries with JPA

JPQL has its roots in the Enterprise JavaBeans Query Language (EJB QL) that was first introduced in the EJB 2.0 specification to allow developers to write portable "find" and "select" methods, for container-managed entity beans. It was based on a small subset of SQL and it introduced a way to navigate across entity relationships both to select data and to filter the results. JPQL was then introduced as part of the JPA to significantly extend EJB QL, thus eliminating many of its weaknesses, while preserving backward compatibility.

Kundera

Kundera is an open source project started by **Impetus Inc.** an India based tech company active in Big Data and Cloud engineering. Kundera provides

a JPA 2.1 compliant object-datastore mapping library for NoSQL datastores leveraging the existing NoSQL database libraries, and builds on top of them a wrapper compliant to the JPA specifications.
The main advantage of the Kundera approach is that, using a well known and defined interface, developers do not need to learn a new framework, furthermore, the use of the JPA interface permits code re-usability since each annotated entity and each JPQL query will work independently from the underlying technology actually used.

Spring-data

Spring Data [10] is a high level **SpringSource** project whose purpose is to unify and ease the access to different kinds of persistence stores, both relational database systems and NoSQL data stores. It is an umbrella project which contains many sub-projects that are specific to a given database. The database currently supported are: MongoDB, Redis, Neo4j, CouchBase, Elasticsearch, Cassandra, DynamoDB and JDBC support.
JPA introduced a standard for object/relational mapping (i.e. mapping object graphs to relational database tables), with Spring Data, this support is extended to NoSQL datastores with object-like data structures. Each type of datastore comes with its own set of annotations that provides the needed meta information for the mapping. An example of such diversity, in handling different datastore mapping, is reported in the code 2.3 which shows the annotations required to be able to persist correctly the same entity in MongoDB and in Neo4j. As can be seen, the approach does not permit the migration form a NoSQL technology to another, without modifying the application code due to the very different requirements in terms of class and fields annotations.

```
1  // MongoDB mapping
2  @Document(collection="usr")
3  public class User {
4      @Id private String id;
5      @Field("fn") private String name;
6      private Date lastLogin;
7      ...
8  }
9
10 // Neo4j mapping
11 @NodeEntity
```

```
12 public class User {
13     @GraphId Long id;
14     private String name;
15     private Date lastLogin;
16     ...
17 }
```

Listing 2.3: Spring Data object mapping

When working with data, developers generally write some Data Access Object (DAO) classes that encloses the required logic for implementing CRUD operations or build queries. With Spring Data, DAO classes are completely handled by the framework, requiring the user only to provide an interface of the DAO that extends a specific Spring Data repository which will map the operation to the underlying database specific implementation. An example of this is reported in the code 2.4.

```
1 public interface UserRepository extends MongoRepository<User, String> {
2     List<User> findByName(String name);
3     List<User> findByEmail(String email);
4 }
```

Listing 2.4: Spring Data repositories

PlayORM

PlayORM [8] is an open-source library developed by **Buffalo Software** with the aim of speeding up developer productivity of developing applications which interfaces with NoSQL databases. Currently supports Cassandra, MongoDB and HBase.
PlayORM takes great inspiration from the JPA interface, but it recognizes that the JPA was designed for RDBMS and thus they have re-defined the JPA interface for better cope NoSQL databases. The framework makes use of some JPA interfaces, such as `EntityManager`, for CRUD operations, and the `Query` interface, for queries, but it re-define all the annotations. Furthermore it defines an extensions of JPQL, called S-JQL (which stands for Scalable JQL), that adds to JPQL the keyword `PARTITIONS` and which allows the user to specify the specific data partition on which to execute the query.
An example of entity defined with PlayORM is shown in the code snippet 2.5 and it shows the great similarities with the JPA approach.

```
1 @NoSqlEntity
2 public class Employee {
3     @NoSqlId
4     private String id;
5     private String lastName;
6     @OneToOne
7     private Phone phone;
8     ...
9 }
```

Listing 2.5: PlayORM object mapping

Apache Gora

The aim of Apache Gora [7] is to extend the concept of Object Relational Mapping tools (ORM) to introduce Object-to-Datastore Mapping where the underlying technological implementations rely mostly on non-relational data models. In essence Gora provides a storage abstraction for NoSQL technologies. Gora thus gives the user an easy-to-use in-memory data model and persistence for big data framework with data store specific mappings and built in Apache Hadoop support.

The objectives of Gora can be grouped as follows:

- **Data Persistence**: persisting objects to Column-based stores such as Apache HBase, Apache Cassandra, Hypertable; key-value stores such as Voldermort, Redis, etc; SQL databases, such as MySQL, HSQLDB, flat files in local file system of Hadoop HDFS;
- **Data Access**: an easy to use Java-friendly common API for accessing the data regardless of its location;
- **Analysis**: accessing the data and making analysis through adapters for Apache Pig, Apache Hive and Cascading;
- **MapReduce support**: out-of-the-box and extensive MapReduce (Apache Hadoop) support for data in HDFS.

2.4 Cloud Platform Independent Model

Cloud Platform Independent Model (CPIM) [14] is a Java library built in order to make Java developers able to abstract their application logic from the specific PaaS Provider on which the application will actually be deployed.
During the life cycle of the application may be necessary, for example, due to changes in application requirements or in the business strategy, to move the application to a different cloud provider. In this process, the application needs to be re-engineered since, even if services are similar among various providers, they expose different API, locking the application to the specific PaaS environment; this problem is commonly referred to as vendor lock-in.
The aim of CPIM is to overcome the vendor lock-in that affect the current PaaS industry by providing, to application developers, a common interface to interacts with many cloud services. The library then, at run-time, maps the methods invocations on the generic interface, to specific cloud provider method invocation.
The library now support three different cloud providers: Google App Engine, Microsoft Azure and Amazon AWS. The services that are supported through a common interface are: the blob storage, the mail service, the memcache service, the SQL service (MySQL for Google App Engine and Amazon AWS while SQL Server is the supported solution for Microsoft Azure), message queues service and NoSQL service (Google Datastore for Google App Engine, Azure Tables for Azure and Amazon SimpleDB for Amazon AWS).

2.5 Summary

This chapter introduced some of the main reasons that lead to the NoSQL database introduction and why the industry is so interested in those kind of technologies. We presented the main projects that have born trying to define a standard NoSQL language or a standard way to communicate with different NoSQL databases, and we gave a quick overview of the choices made by each product. Finally it was presented an overview of the JPA interface and the CPIM library, a more general approach for a common language definition in PaaS environments.

Chapter 3

Problem setting

3.1 Introduction

In this chapter we expose the motivations that lead us to conduct this work, in particular, we analyze the current problems in the NoSQL service implementation of the CPIM library and propose a solution to address them and, at the same time, increasing the number of NoSQL database supported by the library. Furthermore, we will discuss why we decided to include the possibility for the CPIM library users to be able to migrate and synchronize data, across databases, by means of a migration system called *Hegira*.

3.2 CPIM NoSQL service

The CPIM library uses various implementation of the JPA interface to ease the communication with different NoSQL databases:

- Google Datastore is supported by means of the Google JPA implementation around Datastore API;
- Azure Tables is supported through *jpa4azure* a third party implementation of the JPA interface for Tables;
- Amazon Simple DB is supported through *simpleJPA* a third party implementation of the JPA interface for Simple DB.

By choosing the cloud provider inside the *configuration.xml*, the library knows, at run-time, which interface should be used for the service and this holds also

for the NoSQL service. Hence to use Google Datastore as NoSQL database, Google must be selected as cloud provider.
The aim of CPIM is to offer to the user a way of writing cloud application in a provider-independent fashion, to be able to migrate the application from a provider to another without the necessity of re-engineer the application. For the NoSQL service this is achieved by meas of the JPA interface that, beside the fact that is is not a standard for accessing NoSQLs, many projects came into play trying to bring the benefit of the JPA interface also in the NoSQL world.

As discussed in chapter 2, the use of the JPA interface in defining a common interface for accessing NoSQL databases is a solution widely adopted, thus the choice of using the JPA as abstraction layer for the NoSQL service in the CPIM is a valuable choice. However the current implementation have significant problem:

P.1 the application code written to interact with the NoSQL service is not interoperable and thus, the user is required to modify the application code in order to be able to move the application to a different cloud provider.

Moreover the NoSQL service suffer of some limitations too:

L.1 the choice of the NoSQL database is strictly bind to the selected cloud provider;

L.2 even if the selection of the NoSQL database would be possible, the number of supported NoSQL database is very limited.

For **P.1**, the problem reside in the fact that for each of the currently supported database, has been found and integrated into CPIM, a specific implementation of the JPA interface. Even through JPA is a well defined standard, not every JPA provider follows strictly the specification and thus, different provider can behave differently while persisting the same entities, since they interpret differently the semantic of some JPA annotation.
An example of this problem is how Collection fields are currently handled in CPIM. In the Google JPA implementation for Datastore and in the JPA

implementation for Amazon SimpleDB, Collection fields are handled correctly, with respect to the JPA specification, thought the `@ElementCollection` annotation, while, in the JPA implementation for Azure Tables, Collection fields needs to be annotated with the `@Embedded` annotation. This require a modification of the code and thus eliminates the effort of CPIM in achieving code portability among PaaS.

As regards **L.1** and **L.2**, we would like to give to the user the ability to persist data in the database that best fit his requirements. For example if the user application will generate data that should be processed with Hadoop, the best solution is to store those data in an HBase instance since its integrate easily in Hadoop. Therefore we want to make the user able to persist different entities in different datastore based on his needs and without the limitation of a specific NoSQL technology.

3.2.1 Proposed solution

The proposed solution is mainly about the integration of Kundera, a JPA compliant ORM for NoSQL databases, as unique persistence layer for the NoSQL service. This integration will be useful to solve the problems and mitigte the limitation outlined as follow:

- since Kundera will be the unique persistence provider for the library we will relay only on one implementation of the JPA interface overtaking in this way, the problem **P.1**, related to different interpretation of the JPA annotation, and thus achieving complete portability of the code of the application model since no code modifications are required to work with different NoSQL database through Kundera;

- the integration of Kundera permits a redesign of the NoSQL service aimed to decouple the chosen PaaS provider and the NoSQL technology overcoming limitation **L.1**, by giving to the user the ability of deciding which technology is more suitable for his needs. Furthermore exploiting the polyglot-persistence provided by Kundera, the user will be able to persist entities within different NoSQL databases at the same time, sim-

ply by defining accordingly the persistence unit in the *persistence.xml* file;

- choosing Kundera as persistence layer we can actually take advantage of the already developed extension for many different NoSQL databases, adding as a result the support of those database to CPIM, and thus overcoming the limitation **L.2**.

There are many reasons why we choose to use Kundera as persistence provider for the NoSQL service of the CPIM library. The main reason is that Kundera, through the use of the JPA interface will permit to the user to handle the complexity of NoSQL databases with expertise he already uses for SQL systems. Furthermore Kundera is in the field from 2010 and thus have a big and active community, built in many years of activity, and has been used successfully in some production environment.
The only drawback is that Kundera does not support any of the NoSQL datastore currently supported by CPIM. Fortunately Kundera have, as its primary goal, to make the library as much extensible as possible, to let developers build their own client around new NoSQL technologies. The solution will thus be to develop the needed extensions for Kundera.

The work on the CPIM NoSQL service will thus require:

- the integration of Kundera as the unique persistence provider in the NoSQL service of CPIM;
- the development of two brand new Kundera extensions, one for Google Datastore and one for Azure Tables.

3.3 Hegira integration

NoSQL technologies do not offer a common querying language to interact with them, as SQL does for RDBMS. Furthermore, NoSQLs offer a simpler interface with respect to RDBMS and each of them exposes a proprietary API tailored to the specific database needs. This requires to interact with NoSQL databases at a lower level of abstraction, moving a good amount of developing effort toward the user. Given the amount of NoSQL solutions available nowadays

[3], and this low-level approach in using such technologies, a company that want to adopt a NoSQL solution to manage its data, finds itself locked to the chosen technology. For this reasons while NoSQL solutions can be appetible to industry, the high costs of application re-engineering and the necessity of investments on qualified personnel, disrupt the adoption of such technologies. The CPIM library can mitigate this vendor lock-in problem by giving to the user the freedom to choose the NoSQL solution that best fit its application requirements and, furthermore, by using the JPA interface, gives the possibility to interacts with NoSQL databases with expertise that companies already have. However when a company actually faces the problem of changing the storage solution for its data, even if the application, through frameworks like CPIM, permits effortless code portability, data migration from the old storage to the new one became a huge problem.

Data migration has became a key feature in modern IT, there exists many reasons to move data from one storage to another: for load balancing, system expansion, failure recovery, etc.
Typical migration solutions involve applications stop to move the data offline and restart the application when the process has been completed, to guarantee the correctness. On the other hand, modern computer systems are expected to be up continuously and thus even planned downtime to accomplish system reconfiguration is becoming unacceptable [20].

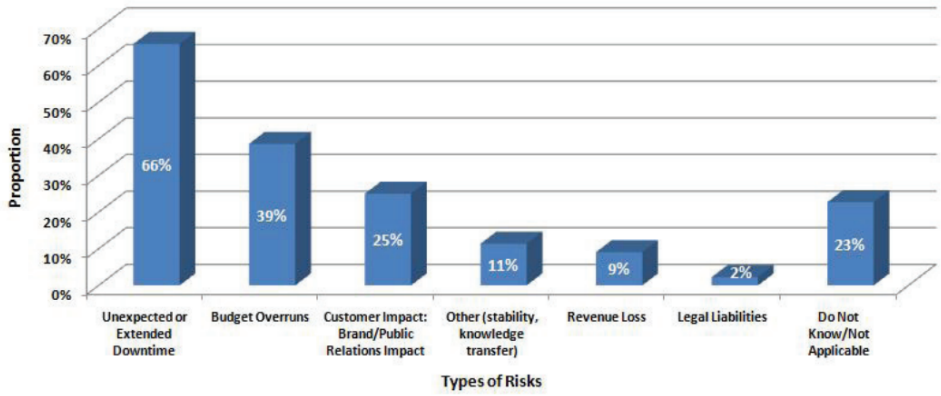


Figure 3.1: Perceived risk in data migration [20]

To mitigate those problems we want to extend the CPIM library to make it

able to interact with *Hegira* a migration system able to perform interoperable data migration and synchronization across column-based NoSQL databases [12]. *Hegira* is already able to migrate data offline, but in many cases this solution is not acceptable since this requires to turn off the application for a period of time that depends on the volume of data that needs to be migrated towards the new database. Downtime costs and risks of data loss can be problematic so, *Hegira* was extended to be able to perform a live-migration of the data by keeping them synchronized on the source and the destination database. This feature needs to be exploited at application level and thus we decided to embed it inside the CPIM NoSQL service, in order to make it as transparent as possible to the user.
The CPIM library needs to be aware of the state of both the synchronization and migration systems and acts accordingly intercepting user operation and sending data manipulation queries (DMQ) to the migration system which is in charge of keeping the data consistent across the replicated databases.

Chapter 4

Kundera extension

4.1 Introduction

This chapter briefly presents in section 4.2 Kundera modular architecture, the way in which Kundera is supposed to be extended, the problems occurred in the process and how the community helped in achieving the result.
In section 4.3 are discussed the detail of the two developed Kundera extension, in particular section 4.3.1 describe the extension for Google Datastore while section 4.3.2 the one for Azure Tables.

4.2 Overview of Kundera

Kundera [5] is an implementation of the JPA interface that currently supports various NoSQL datastore. It supports by itself cross-datastore persistence in the sense that its allows an application to store and fetch data from different datastores. Kundera provides all the code necessary to implement the JPA 2.1 standard interface, independently from the underlying NoSQL database which is being used.

Currently supported NoSQL databases are:

- Oracle NoSQL (versions 2.0.26 and 3.0.5)
- HBase (version 0.96)
- MongoDB (version 2.6.3)

- Cassandra(versions 1.2.9 and 2.0.4)
- Redis (version 2.8.5)
- Neo4j (version 1.8.1)
- CouchDB (version 1.0.4)
- Elastic Search (version 1.4.2)

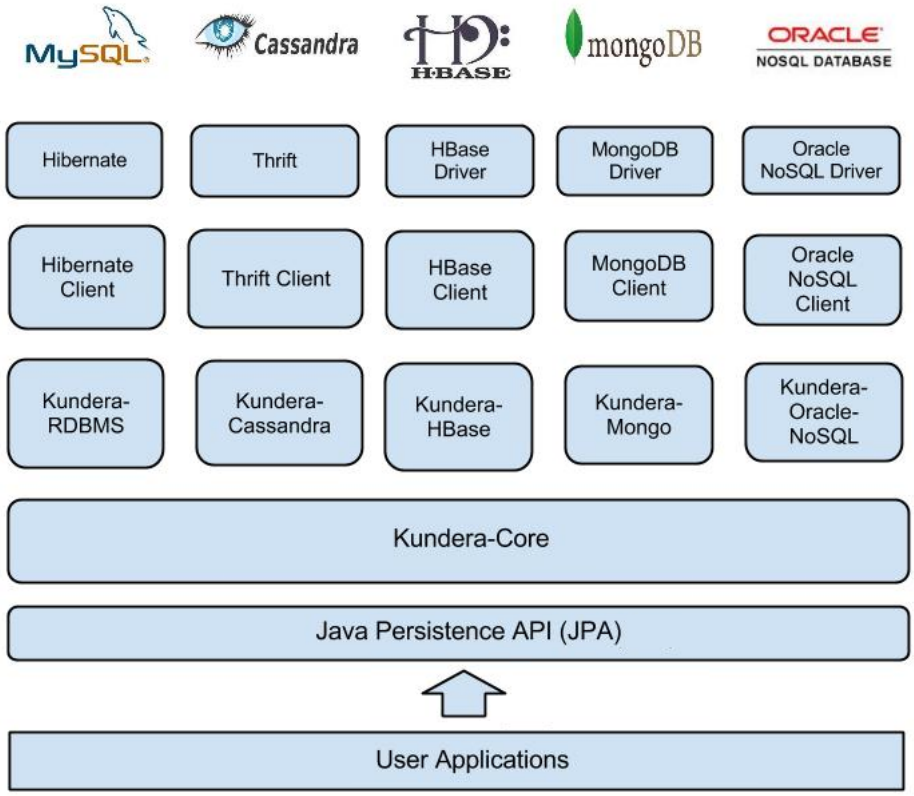


Figure 4.1: Kundera architecture [5]

The architecture of Kundera is shown in Figure 4.1. The figure highlights the fact that the user application interacts with Kundera simply by exploiting the standard JPA interface implemented in the Kundera-Core.
Kundera-Core, each time an operation need to be executed on the underlying database, delegates the operation to the appropriate `Client` (creating it through a `ClientFactory` if it does not exists yet). Clients are then responsible of actually executing the operation on the underlying database.

4.2.1 Kundera's Client Extension Framework

Kundera tries to offer a common way to interact with NoSQL databases through a well defined interface furthermore, since it is as on open source project, it makes other developers able to extend it, adding support for other databases. The *Client Extension Framework*, described in the Kundera documentation, provides a short description about how Kundera clients should work and gives a description of interfaces and classes that should be developed in order to make the client work properly.

Basically to build a new Kundera client, these are the blocks to be developed:

- the `Client`, which is the gateway to CRUD operations on database, except for JPQL queries and named queries;
- the `ClientFactory`, which is used by Kundera to instantiate the new Client;
- the `QueryImplementor`, which is used by Kundera to run JPA queries by invoking appropriate methods in Entity Readers;
- the `EntityReader`, which is used by Kundera to translate the queries into correct client method calls;
- optionally the `SchemaManager`, to support automatic schema generation.

4.2.2 Approaching the extension

While trying to extend Kundera we faced several problems that were not covered by the documentation, two were the main problem in understating what to do and how:

- when actually defined the classes and implemented the interfaces, it turns out that there are actually little differences both on interfaces and the required methods;
- the documentation lacks completely in describing what kind of information are carried by the argument of the methods that needs to be implemented.

After getting the updated information from the community it turns out that the `Entity Reader` was unnecessary and all the translation, from JPA queries to datastore specific queries, and their executions, should be done in the `QueryImplementor`. Unfortunately no help was given by the community about the issue on methods arguments. Hence the most valid solution to approach the development of the extension was in a test driven way trying so to reverse engineer those arguments.

4.3 Developing client extensions

The work carried out has also focused on the development of two Kundera extensions, the first one for Google Datastore and the second one for Azure Tables.

Kundera *Client Extension Framework* provides a generic interface which methods are supposed to carry out a lot amount of code. An example of this is the persist operation that is handled by the `onPersist` method. Besides actually perform the persist operation, it have to create an object that can be persisted in the specific database by reading all the entity meta-data given as arguments, looking for example to relational fields. The adopted solution is a template pattern in which each method maintains the main algorithm structure and delegates every operation to a specific hook method. An example of this approach for the Datastore case, is reported in pseudo code in the snippet 4.1.

```
1 @Override
2 protected void onPersist(...) {
3     Entity gaeEntity = DatastoreUtils.createEntity(entityMetadata, id);
4     handleAttributes(gaeEntity, entityAttributes);
5     handleRelations(gaeEntity, relationHolders);
6     handleDiscriminatorColumn(gaeEntity, entityMetadata);
7     performPersist(gaeEntity);
8 }
```

Listing 4.1: Template for the persist operation

For the Azure Tables extension, since it has been developed as the last one, the same structure has been kept and so it was only necessary to update the code of the hook methods.

The following sections presents these extensions separately, describing each one of the supported features.

4.3.1 Google App Engine Datastore client

Google App Engine Datastore [2] is the NoSQL solution build on top of Google BigTable a sparse, distributed, persistent multidimensional sorted map available in the App Engine platform [13].

JPA identifier

The most basic unit that can be stored in Google Datastore is an *Entity*, which is identified by a *Key* and it is composed of *Properties*. Keys contain various information about the entity itself:

- the entity *Kind*, which is used to group entities of the same type;
- an entity identifier, used to distinguish entities of the same type;
- an optional parent entity .

Inspired by the Google JPA implementation for Datastore [4] the idea was to use the Java class representing the datastore *Key* as identifier for the entity, but, unfortunately, this was not possible since Kundera support only a predefined defined set of Java data types.
Hence the adopted solution is to handle the key internally. Each time an operation on Datastore is required the key, relative to the entity, is built. The *Kind* is directly mapped to the table name and the Key identifier is the user defined id specified in the `@Id` annotation. The `@Id` annotation, in fact, is the annotation (available in the JPA specification) that is used to identify the class field that will be used as primary key in the resulting table on the underlying database.

IDs can be specified by the user or automatically generated, and they can be associated to three different data types

- `@Id` annotation on a `String` type field
- `@Id` annotation on a `Long` type field
- `@Id` annotation on a primitive `long` type field

Kundera extension

Since Kundera supports the JPA feature for auto-generated IDs by using the annotation `@GeneratedValue`, this possibility has been exploited also for the Datastore extension and so the user can annotate a `String` ID field so as that it will be auto-generated and its value will be a string representation of a random java `UUID`.
Auto-generated IDs are supported by Kundera only with `AUTO` or `TABLE` strategy, it was not possible to use the Datastore API to generate IDs since it is necessary to know the *Kind* of the entity to be persisted but neither the `AUTO` strategy nor the `TABLE` one provides this information at generation time.

Consistency

In Datastore entities are organized in *Entity Groups* based on their *Ancestor Path*. The Ancestor Path is a hierarchy of entities whose keys have relation among themselves.
Consistency is managed through entity groups and so by defining the ancestor paths. Entities within the same entity group are managed in a strongly consistent means. Entities which are not in an entity group are treated with eventual consistent policy.
Datastore allows to create ancestor paths by defining entities parental relationships between entities and is it a task left to the user. Datastore low-level API also leave this task to the user, for example in Objectify [18], a wrapper for these API, the developer makes use of a `@Parent` annotation to make the user able to specify the parent relationships and hence to be able to organize entities through the ancestor path.
Since JPA is a well defined standard, adding such kind of annotation will break the standard and the only alternative left is trying to automatically guess the ancestor path.
An approach to do so can be to look at JPA relationships since they are clearly a good place to found information for guessing if two entity kind can be hierarchically related, hence for each type of relation we may define some solutions that can be adopted:

- for **One to Many** and **One to One** relationships, since there is an owning side of the relationship, the owning entity can be used as parent for every related entity.

- **Many to One** relationships can be considered as **One to Many** relationships.

- as regards **Many to Many** relationships, a solution can be to persist the elements of the join table as child of the entity in the owning side of the relationship but the specular solution (persist elements as child of the entity on the non-owning side) can be adopted too. The only solution that is not acceptable in this case is to persist both the entities, the one on the owning side and the one on the non-owning side, as parent to an element of the join table. This is principally due to the fact that this will require, as pointed out later, the Key of the join table element to be able to retrieve the entities from Datastore.

Even though it could have been possible to infer such relationships we choose not to implement it inside the Kundera extension for two main reasons:

1. entities are not require to have a single relationship so, for example, if an entity contains two different relationships of type *One to One* there is no way to decide which one should be used, unless asking to the user;

2. entities with a parent require, beside their own Key, the parent Key (and thus its Kind and identified) to be universally identified. For how Kundera is structured those information are not available and even if the parent entity Kind can be retrieved from Kundera meta-data (by searching in the relationships meta-data), its identifier is not available inside meta-data as thus, since the complete Ancestor Path cannot be built, the entity cannot be retrieved.

For those reasons it was not possible to automatically guess ancestor paths by means of JPA relationships or make the user able manage them directly through a specific annotation without causing errors. Each Kind is persisted as a root Kind hence each entity is stored as a separated entity group identified by its own Kind (the name of the JPA table associated to the entity).

JPA relationships

All the JPA supported relationships have been implemented in the client as it would have been done in a relational database. So for **One to One** and **One**

to Many relationships, on the owning side of the relationship, a *reference* to the non-owning side entity is saved.
For **Many to One** relationships there would be two solutions:

- to persist a list of *references* to the related entities;
- not to persist anything within the entity and fill the relationship with a query.

The second solution has been adopted since it is more consistent with the other Kundera client implementations and with the classic implementations in relational databases.
As regards **Many to Many** relationships a join table is created based on user directives specified by means of the entity class annotations. The join table is filled each time a many to many related entity is persisted and a new *row* is created inside the join table with the *references* to the entities involved in the relationship.
The so far called *reference* for Datastore is exploited by persisting within the entity the Key (Kind and identifier) of the related entity.

Can be useful at this point to show how an entity, annotated with the JPA standard, is then mapped to a datastore entity. Let's take as example the case described in the code 4.2, the `Employee` class is annotated with many JPA annotations.

- the `@Entity` annotation specify to the JPA provider that this will be mapped to an entity in the underlying database and the `@Table` annotation specify the name that the table should have and the persistence unit to which the entity refer;
- the `id` field will handle the identifier for this entity as it is annotated with the `@Id` annotation and furthermore this id will be auto-generated due to the presence of the `@GeneratedValue` annotation;
- the `@Column` annotations specify to the JPA provider under which name the fields should be persisted on the underlying database.

The same is for the `Phone` entity which is related to `Employee` with a one to one relationships and thus a *reference* of the related phone entity will be persisted within the employee one.
The resulting entities on Datastore will be persisted as shown in table 4.1.

```
1  @Entity
2  @Table(name = "EMPLOYEE", schema = "keyspace@pu")
3  public class Employee {
4      @Id
5      @GeneratedValue(strategy = GenerationType.AUTO)
6      @Column(name = "EMPLOYEE_ID")
7      private String id;
8
9      @Column(name = "NAME")
10     private String name;
11
12     /* an employee have one and only one phone */
13     @OneToOne
14     @JoinColumn(name = "PHONE_ID")
15     private Phone phone;
16 }
17
18 @Entity
19 @Table(name = "Phone", schema = "keyspace@pu")
20 public class Phone {
21     @Id
22     @GeneratedValue(strategy = GenerationType.AUTO)
23     @Column(name = "PHONE_ID")
24     private String id;
25
26     @Column(name = "NUMBER")
27     private Long number;
28 }
```

Listing 4.2: Example entities

Key	ID/Name	NUMBER
PHONE:3cb26744	**3cb26744**	123456789

(a) PHONE

Key	ID/Name	NAME	PHONE_ID
EMPLOYEE:112b18e7	**112b18e7**	Fabio	**PHONE:3cb26744**

(b) EMPLOYEE

Table 4.1: Mapping of entity fields on Google Datastore

Queries

Kundera queries are to be expressed in JPQL; the standard JPA query language, which is a object oriented query language based on SQL [15]. Kundera supports all of the clauses of JPQL, but with some restrictions since clauses can be applied only on primary key attributes (the ones annotated with the `@Id` annotation) and column attributes (the ones annotated with the `@Column` annotation).
Once a JPQL query is parsed and validated by Kundera it is passed to the `Query Implementor` together with some meta-data extracted from it which then need to be read in order to build a database specific compatible query.

Google Datastore has on its own a very good support to queries so almost all the clauses are supported except for the *LIKE* one.
To be able to execute queries on properties, Datastore needs to construct secondary indexes for those properties. Those indexes consume the App Engine application quotas to be stored and maintained. The API provides the possibility to decide which property should be indexed, by calling a different method when adding the property to an entity; in fact, `setProperty(String name, Object value)` method is used to set a property which will be automatically indexed, where `setUnindexedProperty(String name, Object value)` can be used to create a non-indexed property.
Since a discriminator is needed to choose between the two methods, other wrappers around Google Datastore low-level API (such Objectify [18]) provide to the user an `@Index` annotation to be placed upon the field that needs to be indexed, but as previously explained, it is not convenient to add other annotation to the JPA standard since this will break interoperability. For those reasons, and in order to be able to actually execute queries, all properties are set as indexed. This choice make queries able to be executed upon every property of an entity but this, as stated before, requires App Engine to maintains secondary indexes, consuming application quota.

Table 4.2 shows a complete list of the Kunedra supported JPQL clauses and their support for both the developed extensions.

JPA-QL Clause	Datastore support	Tables support
Projections	✓	✓
SELECT	✓	✓
UPDATE	✓	✓
DELETE	✓	✓
ORDER BY	✓	✗
AND	✓	✓
OR	✓	✓
BETWEEN	✓	✓
LIKE	✗	✗
IN	✓	✗
=	✓	✓
>	✓	✓
<	✓	✓
>=	✓	✓
<=	✓	✓

Table 4.2: JPQL clauses support for the developed extension

Embeddable Classes

Embeddable classes are user defined persistable classes that function as value types. As with other non entity types, instances of an embeddable class can only be stored in the database as embedded objects, i.e. as part of a containing entity object. A class is declared embeddable by annotating it with the `@Embeddable` annotation and can then be used in an entity, as a value type, annotating the field as embedded with the `@Embedded` annotation.
Implementation of those kind of entities is straightforward for Datastore because the embeddable classes can be mapped to the natively supported `EmbeddedEntity`. The implementation makes use of such feature by translating the embeddable entity into a Datastore embeddable entity and then persisting it within the parent entity.

Collection fields

JPA standard supports collection or maps to be used as entities field by using the annotation `@ElementCollection`.

Java Collections are natively supported by Google Datastore but are supported only if composed of one of the supported Datastore data types which includes the main Java data types, such as `String` and `Long`, and Datastore specific ones, such as `Key`.
To be able to save whatever kind of collection (or map) independently of the data type that composes it, the collection (or map) itself is serialized into a `byte` array when persisted and de-serialized when read. To simplify the development, also Lists of primitive types, even if supported natively, are serialized.

Enum fields

Enum fields are supported by the JPA through the annotation `@Enumerated`, by simply persisting its string representation and, when the entity is read back, by instantiating the corresponding enum type.

Schema Manager

The schema manager, as required by Kundera, has to make use of four operations:

- *validate*, which validates the persisted schema based on the entity definition;
- *update*, which updates the persisted schema based on the entity definition;
- *create*, which creates the schema and, thus the tables, based on the entity definitions;
- *create_drop*, which drops (if it exists) the schema and then re-creates it by re-creating the tables based on the entity definitions.

The first two cases are quite useless for Google Datastore and in general for NoSQL databases, since there is typically no fixed schema for the entities.

Entities with same *Kind* can have different properties without restriction. Also the *create* case is meaningless for Datastore since when a new entity of an unknown *Kind* is persisted it is created without the need of explicitly defining it first as a new *Kind*.
The last case *create_drop* will just drop the current schema, deleting all the persisted kinds and so all the related entities, without re-creating the schema since it constructs by itself.

Datastore specific properties

Kundera offers the possibility to define some datastore specific properties in an external XML file that needs to follow a simple structure. This file is referenced inside the `persistence.xml` and it is optional.

This possibility is exploited by the Datastore extension and make the user able to configure the following properties:

- `datastore.policy.read`, to set the read policy;
- `datastore.deadline`, to define the RPCs calls deadline;
- `datastore.policy.transaction`, to specify if Datastore has to issue implicit transactions.

Those properties are read by the `Client Factory` and used to initialize the datastore connection with the required parameters.

A complete reference of Google Datastore extension configuration is available in the the appendix A.3.

4.3.2 Azure Table client

Azure Tables [1] is the NoSQL solution developed by Microsoft, it is a key-value storage and it is available inside Azure environment.

JPA identifier

In Azure Tables an entity to be persisted must either implement a special interface `TableServiceEntity` or be translated into a `DynamicEntity` which

is basically a dynamic property container (i.e. it does not impose a fixed data scheme). An entity is then uniquely identified inside a table by a *partition-key* and a *row-key*. Partition keys are used to handle consistency, strong consistency is guaranteed for entities which are stored within the same table and having the same partition key, otherwise consistency will be eventual by default.
Since both partition-key and row-key support only the data type `String` and since the JPA annotation `@Id` can be declared only on one field per class, the partition-key and the row-key are concatenated in a single `String` field and handled internally by the extension through the class `AzureTableKey` (a custom class built *ad hoc* for Azure Tables, since there is no such a class that encapsulate both the partition-key and the row-key). This way the user has complete control over partition-key and row-key and thus on the consistency mechanism.

The user can handle those identifiers in three different ways:

1. manually define the row-key and the partition-key;
2. manually define only the row-key;
3. let the extension handle completely the identifier, annotating the ID field also with `@GeneratedValue(strategy = GenerationType.AUTO)` annotation.

In the first case, to help the user in define both the partition-key and the row-key independently by how are handled internally by the extension, a static method `AzureTableKey.asString(String partitionKey, String rowKey)` is provided; its usage is not required, but in case the ID is manually specified, it must follow the convention used internally by the extension which is `partitionKey_rowKey`.
To be able to specify only the row key, while keeping the partition key set to the default value (which can be modified in the datastore specific property file described later on), to have a more fluent API, an utility method is provided: `AzureTableKey.asString(String rowKey)`
The third and last method will generate a java random `UUID` for the row key and set the partition key to the default value.

JPA relationships

Also for Azure Tables extension, relationships are implemented similarly to relational systems as described previously for Google Datastore in section 4.3.1. In Azure Tables to universally identity an entity the partition-key, the row-key and the table name are required. Since the table name is always provided by Kundera (and is available in the entity meta-data), the only required information to identify an entity are the partition-key and the row-key. When two entities are related, the partition-key and the row-key of the related entity are persisted within the entity that owns the relationship.

Taking the same example described for Google Datastore in section 4.3.1 and reported in the code 4.2, the resulting mapping of the entities fields for Azure Tables is the one reported in table 4.3.

Partition Key	**Row Key**	**NUMBER**
DEFAULT	**3cb26744**	123456789

(a) PHONE

Partition Key	**Row Key**	**NAME**	**PHONE_ID**
DEFAULT	**112b18e7**	Fabio	**DEFAULT_3cb26744**

(b) EMPLOYEE

Table 4.3: Mapping of entity fields on Azure Tables

Queries

Supporting queries for Azure Tables was straightforward, the procedure was the same described in 4.3.1 but due to the different operator supported by Tables, beside the *LIKE* clause, also the *IN* and the *ORDER BY* clauses are not supported.

Table 4.2 shows a complete list of the Kunedra supported JPQL clauses and their support for both the developed extensions.

Embeddable Classes

Embeddable classes (described in 4.3.1) are not supported natively by Azure Tables hence the solution adopted is to serialize the field annotated with `@Embedded`, in order to be able to persist it to the storage like a `byte` array and de-serializing it when the entity is read back.

Collection fields

As described for Datastore in section 4.3.1, JPA supports collections, but these are not supported in Azure Tables, even if composed of supported data types. To support collections and maps, the simplest solution is to serialize the entire collection (or map) to a `byte` array when persisting the entity. When reading back from the database, the entity is de-serialized properly.

Enum fields

Enum fields are supported by the JPA through the annotation `@Enumerated`, by simply persisting its string representation and, when the entity is read back, by instantiating the corresponding enum type.

Schema Manager

The Schema manager (as described in section 4.3.1) has also been implemented for Azure Tables and, like Google Datastore, the first two cases are quite useless since there is no fixed data schema and entities, within the same Table, can have different properties without restrictions.
Azure Tables needs that the table in which entities are stored exists before trying to create entities inside of it so, the *create* case simply iterates over all table names and creates them in the database.
For the *create_drop* case, all tables should be dropped (and so all the contained entities) and re-created. The problem here is that tables deletion is performed asynchronously and so there exists an unpredictable amount of time in which the table cannot be re-created since it still exists, even if it is not listed anymore. To overcome this problem two solutions can be adopted:

- catch the `StorageException`, thrown when trying to create the table while it still exists, put the process to sleep for certain amount of time and then try again until it succeeds;
- Do not delete the table itself, but delete all its entities in bulk.

The first method is clearly dangerous since no deadline is given or even guaranteed for the table delete operation. The second solution is actually not so convenient because, even if deletion is performed as a batch operation, both the partition key and row key must be specified and thus one or more queries must be performed over the table to retrieve at least the partition-key and the row-key for each entity in the table; this will require an high number of API calls and thus an high cost of usage.
We have so decided that for the *create_drop* case a drop of all the Tables is performed and then these are re-created even if this can cause the previously mentioned conflict. This option is left as is for testing purposes since in the storage emulator the problem is not showing up because the Tables storage is emulated over a SQL server instance.

Datastore specific properties

As described for Datastore in section 4.3.1, Kundera provides datastore specific properties file that let the user set some specific configuration.
This possibility is supported also for Azure Tables with the following available properties:

- `table.emulator` and `table.emulator.proxy`, to make the user able to test against the local storage emulator on Windows;
- `table.protocol`, to make the user able to decide between *HTTP* or *HTTPS* for storage API RPCs;
- `table.partition.default`, to let the user specify the value for the default partition key.

For a complete reference to Azure Tables extension configuration see the appendix A.4.

4.4 Summary

In this chapter has been introduced in details how the Google Datastore and the Azure Tables Kundera extensions have been developed, the problems encountered during the development, how they have been addressed and the details of the implementation of the two extensions, including the currently supported features.

Chapter 5

CPIM extension

5.1 Introduction

This chapter presents the CPIM library extension. Sections 5.2 and 5.3 describe the previous state of the NoSQL service in the CPIM, the changes made to integrate Kundera as unique persistence provider and the problems faced during the process.
From section 5.4 up to section 5.8 are described the various parts we developed to support *Hegira*, an interoperable data migration sand synchronization system for NoSQL databases [12], describing the supported features and the design choices that have been put in place.

5.2 CPIM architecture

In order to be able to expose a common interface for the multiple services supported by the library, CPIM adopts heavily the factory and singleton patterns. The main access point of the library is the `MF` (Manager Factory) class, a singleton object which is responsible of reading the configuration files and exposing a set of methods that will build instances for the service factories. The initialization is done through a first call to the `MF.getFactory()` method, which reads the configuration files and build an instance of the `CloudMetadata` class; this class will be referenced by all the other services and it contains all the information stored in the configuration files.

The CPIM library is organized in several packages, each of which is responsible of a particular service. Each service exposes a factory class which is invoked through the `MF` factory; the service factory maintains a singleton instance of the provider-specific service implementation which is built, at the first call, based on the configuration available inside the singleton instance of `CloudMetadata`. The result of this process is that with the same method call, based on the configuration file, can be instantiated a different service implementation.

5.2.1 NoSQL service

The architecture of the NoSQL service before this work has been reported in figure 5.1.
To use the service, the first step is to instantiate a `CloudEntityManagerFactory` and, depending on the configuration file, this factory instantiates the vendor specific factory. For example, in case Google is chosen as vendor, the instantiated factory will be `GoogleEntityManagerFactory`. Each provider-specific `EntityManagerFactory` is responsible of instantiating an `EntityManager` which is the gateway to the underlying database. All vendor-specific `EntityManager`(s) implement the common `CloudEntityManager` interface to achieve uniformity in methods and behavior. The various implementation of the `CloudEntityManager` delegates every method call to the vendor-specific persistence provider.

The JPA is not a default language for NoSQL but, due to its wide usage among Java developers, several JPA implementations have been built for various NoSQL databases (both developed by the vendor of the NoSQL storage or by the community). This means that to support the NoSQL service through the JPA interface, an implementation of the JPA interface must be found or developed *ad hoc*. For this reason there were three different persistence providers in the CPIM library, one for each cloud provider:

- for *Google Datastore* an official JPA implementation (available inside the SDK) was used;

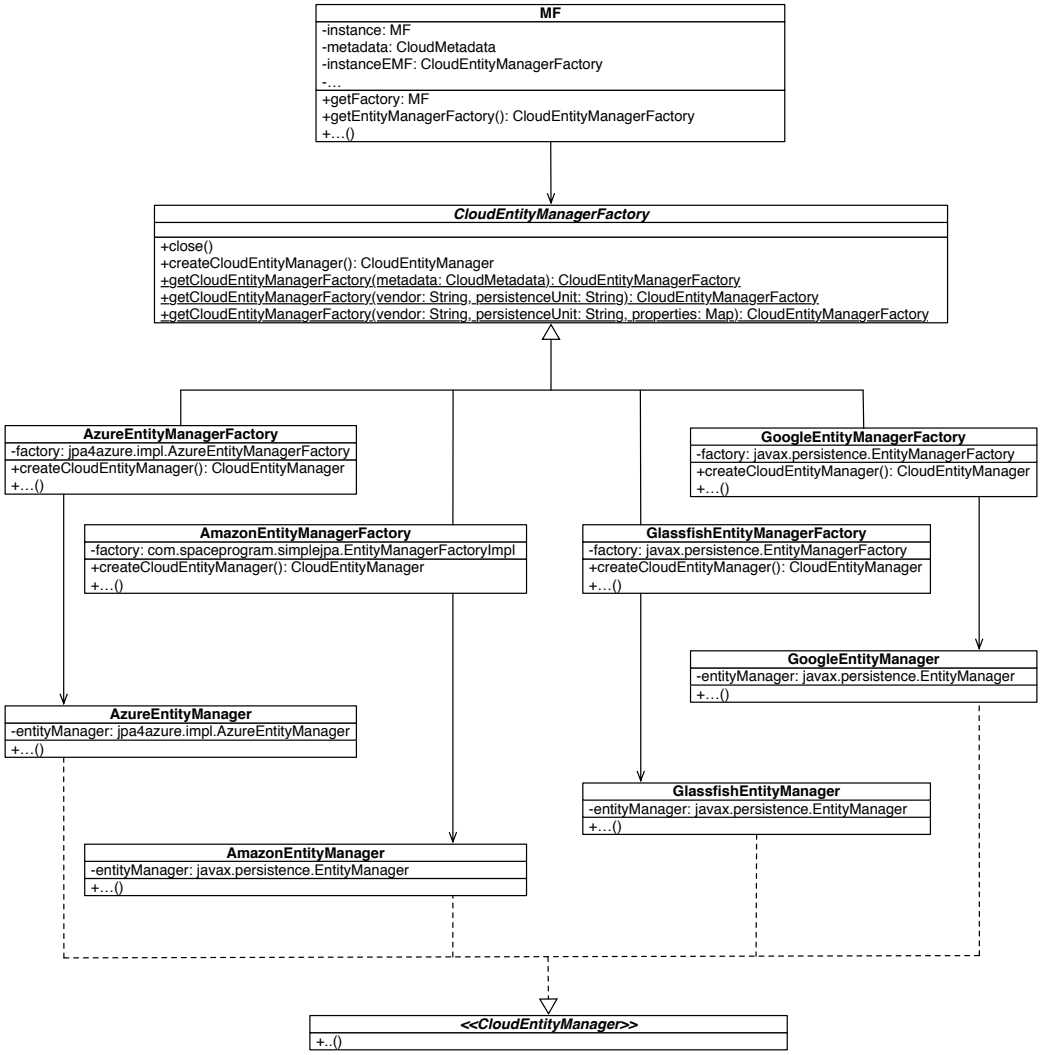


Figure 5.1: NoSQL service architecture

- for *Amazon SimpleDB* it used **SimpleJPA**, a third-party implementation of the JPA interface;
- for *Azure Tables* it used **jpa4azure**, a third-party implementation of the JPA interface.

There are a couple of things to notice: Amazon SimpleDB has been deprecated in favor of DynamoDB and *jpa4azure* is not being maintained anymore, therefore the CPIM library needs to be updated in order to get rid of those outdated software.

5.3 Kundera integration

To solve these problems and reduce the number of software on which the CPIM relies to provide the NoSQL service, the proposed solution is to modify the current CPIM architecture with a unique persistence provider that has been identified in Kundera.

The renewed architecture is resumed in figure 5.2 in which the benefit of having a single JPA provider are clearly visible, the architecture is slightly less articulated and no check on the selected underlying technology is needed since this is handled by Kundera, while reading the *persistence.xml* file, in which the user defines the databases he is interested in. Another benefit of this architecture is that the choice of the NoSQL technology is not bound to the vendor specified in the CPIM configuration file anymore, in fact, it is possible, by configuring the *persistence.xml*, to deploy the application in one of the supported PaaS providers and choose to persist the data in a NoSQL database of another provider. Moreover it is possible to exploit the Kundera polyglot persistency to persist part of the data in a database and another part in another one, defining the persistence units accordingly.

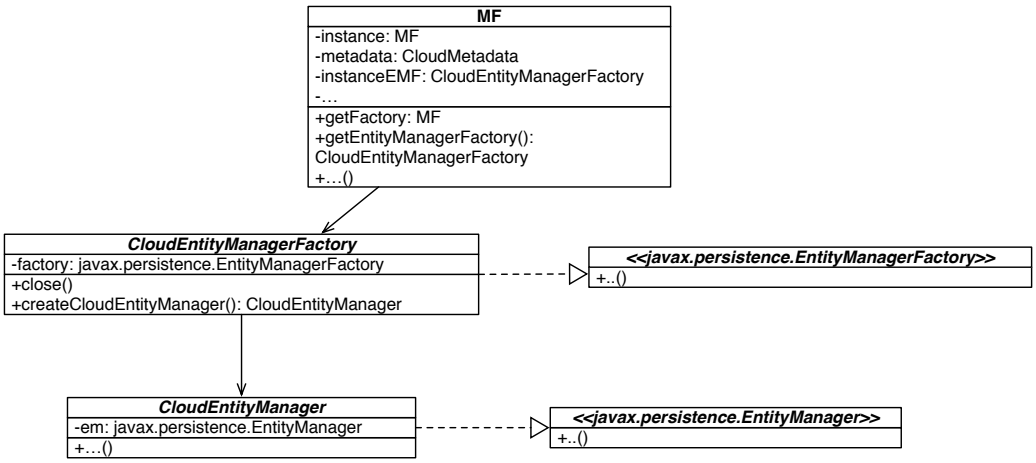


Figure 5.2: The modified NoSQL service architecture

The actual implementation is completely provider agnostic in the sense that actually Kundera is not required as dependency and in fact it is not listed as a dependency for the CPIM. At run-time, when a Kundera client will be listed

among the dependencies of the user application, as well as the CPIM library, the persistence provider dependency will be satisfied.
This provider agnostic implementation is due to the fact that the `CloudEntityManagerFactory` and the `CloudEntityManager` respectively implement the JPA interfaces `EntityManagerFactory` and `EntityManager`. The actual call to the run-time provider is made within the `CloudEntityManager` that, on construction, instantiate an instances of the provider `EntityManager` and uses that reference to delegate every method execution to it.
This can seem an over-designed architecture, but it turns out to be extremely necessary in order to provide a transparent interaction with the migration system, as it will explained later on in this chapter.

5.3.1 Problems encountered

Kundera provides an uniform access through the JPA interface, independently from the NoSQL provider which is being used. The desired target database is defined in the *persistence.xml* file through the `kundera.client.lookup.class` property. For this reason all the old libraries that provide a JPA implementation for a specific vendor can be removed from the CPIM. This tentative of cleaning the dependency of the CPIM caused two main problems:

1. *jpa4azure* turns out to be used also for Queue and Blob service of Windows Azure;

2. Kundera seems to have problems when multiple persistence providers are found in the classpath and currently no way to force the selection of Kundera as persistence provider has been found (besides specifying it in the *persistence.xml* file).

To solve the first problem, the code of the extended version of *jpa4azure* has been inspected. We found that the library was previously extended to support some missing functionalities of the JPA interface and contained two main packages:

- `jpa4azure`, which contained the code that implements the JPA interface;

- `com.windowsazure.samples`, which contains the code to ease the communication with the Azure services.

The `jpa4azure` package has been removed and the library rebuilt since the other package is the one used in the Blob and Queue service. Its possible to completely remove `jpa4azure` but is necessary to rewrite also the CPIM Blob storage service for Azure using the API provided by the Azure SDK.

Removing the `jpa4azure` library caused unexpected errors in CPIM in the code of the Queue service. After some investigations, turns out that, when *jpa4azure* was extended, the class `AzureQueueManagerFactory` were introduced. The problem was that `AzureQueueManagerFactory` make use of the JPA interface to communicate with the Queue service of Azure and thus by removing the support to the JPA interface we have lost the support for Azure Queue service. A solution to this would be to rewrite the CPIM Queue service for Azure, using the API provided by the Azure SDK.

5.4 Hegira integration

To support data synchronization and migration, the NoSQL service was further modified to integrate *Hegira* [17]. An high level schema of the interaction we want to achieve is reported in figure 5.3
In that schema, *dashed* lines represents the normal flow of data from the user application to the local database, the *filled* ones represents the behavior in case a migration is in process.
The CPIM library needs to connect to the migration system in order to understand when a migration is in progress and in that case only bypass the interaction with Kundera (for data manipulation operations) by building a string representation of the user query as a SQL statement. Once this SQL generated string is sent to the *Hegira* commit log that pop the statements and translate them into a datastore-specific operation.

5.4.1 Migration Manager

Interaction with the migration system is handled primarily by the `MigrationManager` class which follows a state pattern represented in the class diagram 5.4.

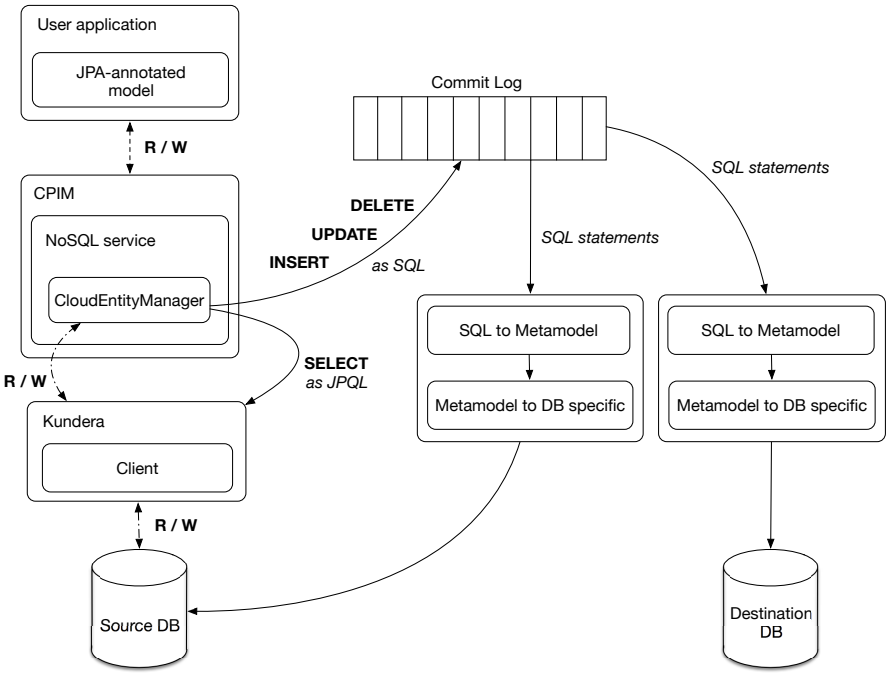


Figure 5.3: High level schema of interaction

The pattern allows to the `MigrationManager` to delegate the method execution to the current state, the state diagram is the one represented in figure 5.5 and is composed by two states `Migration` and `Normal` that encapsulate the required behavior.

5.5 Intercept user operations

The first operation that needs to be analyzed is whether it is possible to intercept user operations in a way that is completely transparent to the user.

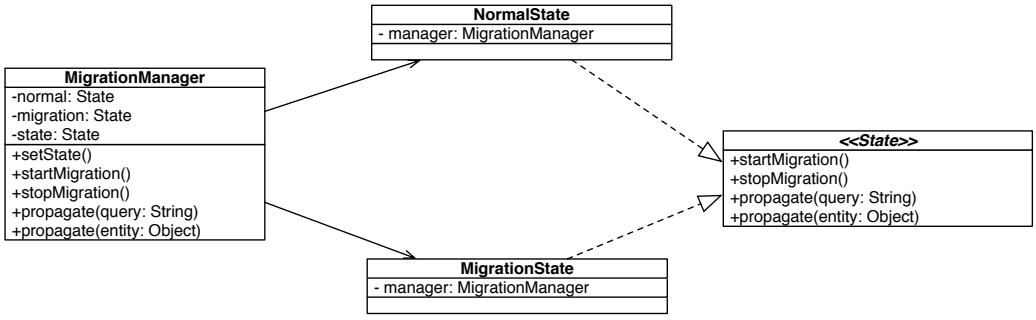


Figure 5.4: `MigrationManager` class diagram

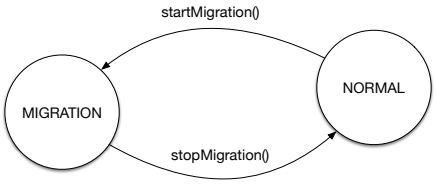


Figure 5.5: `MigrationManager` states

The operations that we want to intercept are the insert, update and delete operations, in that, they are the operations that alter data and, thus, are the ones that the synchronization system needs to process.

5.5.1 Intercepting CRUD operations

CRUD operations are handled by the `EntityManager`, three are the methods that need to be intercepted:

- `EntityManager.persist(Object entity)` for insert operations;
- `EntityManager.merge(Object entity)` for update operations;
- `EntityManager.remove(Object entity)` for delete operations.

The user does not invoke methods directly on the provider entity manager, but he interacts with the persistence provider, through the `CloudEntityManager` class. In the standard implementation (i.e. without the support for the migration system) the `CloudEntityManager`, delegates every method call to the provider entity manager; hence, in order to integrate the migration and synchronization logic, the methods mentioned above should contain the application of logic shown in the snippet of code 5.1, which takes as an example, the update operation.

```
1 public <T> T merge(T entity) {
2     if (MigrationManager.isMigrating()) {
3         MigrationManager.propagate(entity, OperationType.UPDATE);
4         return entity;
5     } else {
6         return delegate.merge(entity);
7     }
8 }
```

Listing 5.1: Integrate migration logic

In case of data migration, the provider is bypassed and a call to the `propagate` method is visible. The call accepts two arguments: the entity to be converted to a SQL statement and the operation that needs to be generated. The method is called on the `MigrationManager` which then delegates the execution to the current state (which should be the migration one). The migration state `propagate` method is responsible for building the requested statements, using the statement builders and then sending the generated statements to *Hegira* commit log. Both actions are described in detail in the following sections.

5.5.2 Intercepting queries

Looking at the JPQL specification [15] it turns out that JPQL does not support *INSERT* statements and so the only way a user has to persist some entities is by means of `EntityManager.persist(Object entity)` method, which was described in the previous section; so, only the remaining queries (*UPDATE* and *DELETE*) need to be intercepted as query.
The JPA interface provides several ways to build and execute queries, all available by calling the proper methods defined in the `EntityManager` interface:

- `createQuery`, which creates a `Query` instance from JPQL query string;
- `createQuery`, which creates a `Query` instance from an instance of `CriteriaQuery`;
- `createNamedQuery`, which creates a `Query` instance from a JPQL query identified by name and declared statically on classes;
- `createNativeQuery`, which creates a `Query` instance from a string representation of the underlying database specific SQL dialect.

Native queries are not supported by Kundera, moreover the migration system tries to abstract from the specific database query language, hence there is no point in supporting this kind of queries. The `createQuery` and `createNamedQuery` methods are supported, instead query creation through `CriteriaQuery` is currently not supported.

The JPA does not provide, through the `Query` interface, a way to get the JPQL representation of the query. Queries are supposed to be written as method

argument when creating them through the `EntityManager` or called by name if they are defined as named queries upon some class. This was actually a problem since, in order to be able to parse the query, its JPQL representation is crucial.
The easiest solution was to implement the interfaces for `Query` and `TypedQuery` respectively with the classes `CloudQuery` and `TypedCloudQuery`.
The wrapping of the persistence provider queries is achieved in the entity manager and it is performed by the query creation method both for the `Query` and `TypedQuery` returned objects. The actual JPA query generation is delegated to the persistence provider; before returning to the user, the result query is wrapped in a `CloudQuery` that contains both the generated query and its string representation.

For named queries things are little trickier since the user create instance of `Query` or `TypedQuery` just by passing the query name. As can be seen from the code snippet 5.2, named queries meta-data are maintained inside the `PersistenceMetadata` class. This class, besides maintaining information about named queries, principally maintains a mapping between table names and their class canonical name (full package plus the class name). The first time this class is queried, its content is created (since it is a singleton instance) and it does not directly read the configuration files, but the `CloudMetadata` instance that has been modified to include all the required parameters that needs to be read from configuration files. The information of table to class mapping is required for statements building and for sequence number handling both described in the following sections.

```
1 public Query createNamedQuery(String name) {
2     String queryString = PersistenceMetadata.getNamedQuery(name);
3     Query queryInstance = delegate.createNamedQuery(name)
4     return new CloudQuery(queryString, queryInstance);
5 }
```

Listing 5.2: Wrap named queries

5.6 Adding support for data synchronization

In this section we report the solution adopted for supporting *Hegira* data synchronization [12], which allows to perform online data migration.
A special look needs to be reserved to the insert operation. When the user updates or delete an entity no matter if through the entity manager or through a query, he already knows the entity identifier, since the insert operation has already persisted that entity into the underlying database and thus generated its identifier. Since we want to support the migration system, the user cannot define its own identifiers, but they need to be assigned from the migration system itself. The main caveats is that, such assignment needs to be made even if the migration is not running yet, so the identifier assignment has to be made in two cases:

1. insert statements built from persist operation during a migration phase;
2. *standard* insert operation through the entity manager during a normal state.

The solution is actually quite simple since everything can be checked inside the `EntityManager.persist` method as described in the snippet of code 5.3.

```
1  public void persist(Object entity) {
2      if (MigrationManager.isMigrating()) {
3          MigrationManager.propagate(entity, OperationType.INSERT);
4      } else {
5          String tableName = ReflectionUtils.getJPATableName(entity);
6          int id = SeqNumberProvider.getNextSequenceNumber(tableName);
7          ReflectionUtils.setEntityId(entity, id);
8          delegate.persist(entity);
9      }
10 }
```

Listing 5.3: Persist operation

In the code snippet is visible a call to the `SeqNumberProvider` class, which is the class responsible of actually interacting with the Hegira component that handle the *sequence numbers* generation i.e., the entities identifiers defined by Hegira to achieve fault-tolerant data migration and synchronization.

Handling the sequence numbers

The sequence numbers are handled by the class `SeqNumberProvider`, a singleton instance that provides a simple way to get the assigned sequence numbers per table. The class diagram of this component, and of the component it interacts with, is shown in figure 5.6.

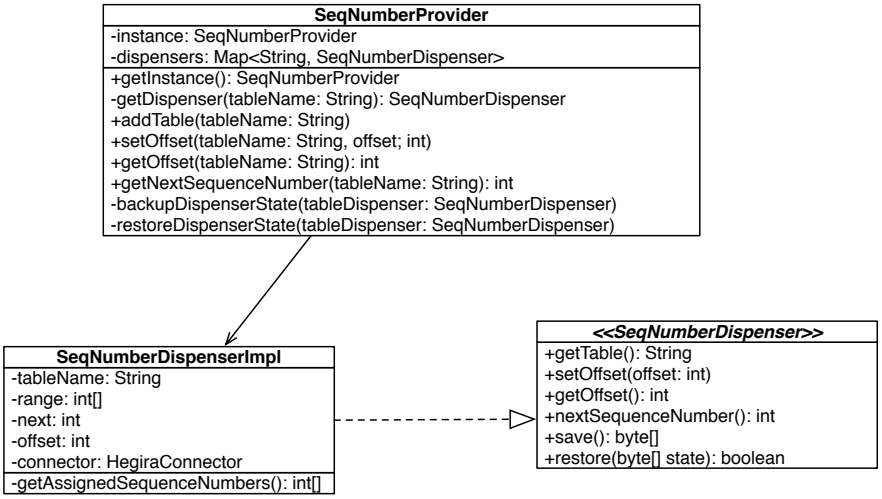


Figure 5.6: Sequence numbers handling architecture

The `SeqNumberProvider` keeps an instance of `SeqNumberDispenser` for each table that needs to be persisted, and it is responsible of:

1. providing a unique access point when requesting the next assigned sequence number for a table;

2. initializing or restoring the state of the dispenser for each table.

The first point is performed through the method `getNextSequenceNumber(String tableName)` that delegates the operation to the correct `SeqNumberDispenser` associated to the requested table. Since `SeqNumberDispenser` is an interface, the actual implementation is delegated to the `SeqNumberDispenserImpl` class. This mechanism has been used to be able to create more dispensers with different logic. The `SeqNumberDispenserImpl` class maintains internally the assigned range of identifiers provided by the synchronization system by specifying the first and the last element of the range. The class consumes, one by one, the identifiers in the range and, when the range has been completed, requests the next range. This mechanism is internally handled, in fact the

`SeqNumberProvider` is only required to call the `getNextSequenceNumber()` method on the dispensers.
The size of the range of sequence numbers requested to the synchronization system, that is used by the `SeqNumberDispenser`, has been made configurable at run-time. A default range size can be set using the *migration.xml* file, otherwise, a call to the `setOffset(String tableName, int offset)` method on the `SeqNumberProvider`, will change, at-runtime, the size of the range of the `SeqNumberDispenser` responsible for the specified table.

The second functionality is achieved by requesting the state representation to the `SeqNumberDispenser`(s) (as a `byte` array), calling the method `save()` on the dispensers and, then, saving it to a Blob storage, or to a file, depending on the configuration specified inside the *migration.xml* file (described in Appendix B). The restoring phase is performed just after construction; if a backup exists either on file or on the Blob storage, the method `restore(byte[] state)` is called on the dispensers, giving them their state representation to restore. In this mechanism, the `SeqNumberProvider` is completely agnostic with respect to the actual state representation chosen by the `SeqNumberDispenser`(s). This makes future extensibility more easy and less constrained.
The list of all the tables to be persisted is retrieved from the `PersistenceMetadata` previously mentioned for named queries.

5.6.1 Contacting the synchronization system

The interaction with the synchronization system has been, until now, only described as a method call. Those calls are made on an external library (`zkWrapper`) which provides an interface to a ZooKeeper instance, in order to communicate with the synchronization system and to receive the assigned sequence numbers. Since the ZooKeeper library uses threads to handle communication, it was not possible to use this library for Google App Engine since, the App Engine run-time, does not permit to spawn thread. The main reasons to communicate with the synchronization system are:

- the migration state listener, that modifies the `MigrationManager` state accordingly;

- the `SeqNumberDispenser`, that needs to retrieve the sequence number assigned to tables.

The adopted solution was to modify the `zkWrapper` library to include an HTTP version that handles the calls not by connecting directly to a zookeeper instance, but contacting a remote server through some defined API, that ultimately interacts with the migration system.

A simple structure has been built to make both the `MigrationManager` and the `SeqNumberDispenser`(s) transparent to the type of client that is used to retrieve information from the synchronization system. The architecture is shown in figure 5.7.

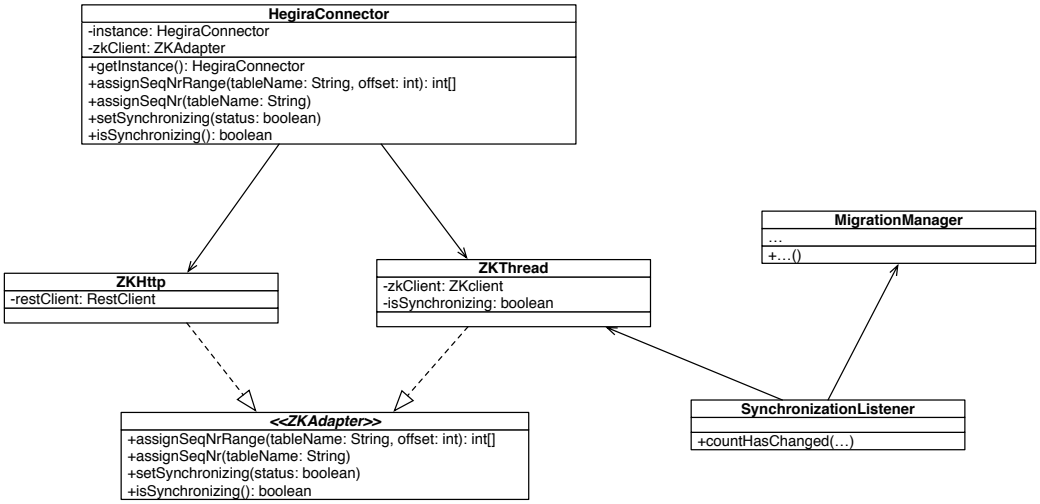


Figure 5.7: Contacting the synchronization system

The `HegiraConnector` is the class responsible of deciding which kind of client needs to be instantiated, the decision is done by reading the configuration that the user specified in the *migration.xml* file, which is parsed by the CPIM and kept in the `CloudMetadata` class. The `HegiraConnector` keeps internally an instance of the chosen client and provides access to its method by delegation. The two available clients implements the interface `ZKAdapter`, built to uniform the methods of the two implementations.

Thread-based client If the user deploies the application on a thread-capable cloud environment (for example on IaaS), and configures the *migration.xml* accordingly, an instance of `ZKThread` is built. This version of the

client directly uses the implementation of the library `zkWrapper` (since there should not be any problem in thread spawning). The `isSynchronizing()` method returns a value which is kept inside the `ZKThread` instance and is queried by the `MigrationManager`. Both the state of the `MigrationManager` and the value inside `ZKThread` are modified by the `SynchronizationListener` which is asynchronously notified by the `zkWrapper` library when the synchronization state changes.

HTTP-based client In the case in which threads are not supported by the cloud provider (for example on PaaS), the client version that is instantiated (by looking at the configuration) is `ZKHttp` which uses the API-caller added to the `zkWrapper` library. Since no listener can be register to be asynchronously notified of a change in the synchronization state, each call of the `MigrationManager` to the method `isSynchronizing()` will perform an API call to the remote server, which will return the state of the synchronization system.

Before focusing on the statements building, it may be useful to visualize the interaction so far presented. An high-level flow chart is reported in figure 5.8.

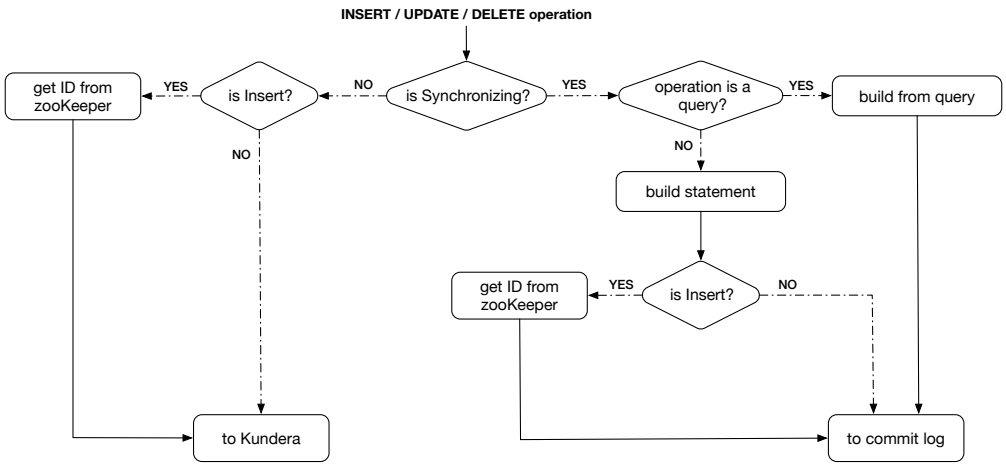


Figure 5.8: Interaction flow chart

5.7 Build statements from user operations

In the previous section we have focused on the sequence number retrieval, in this section we will focus on the generation of the Data manipulation queries (DMQs) to be sent to *Hegira* commit-log.

In order to be able to create SQL-like statements both from queries and from objects, the first step has been to introduce the *statement* concept in the CPIM library. This has been done through the abstract class `Statement` that encapsulate the structure needed for maintaining the necessary data for the statements. The `Statement` class is then extended by the three classes: `InsertStatement`, `UpdateStatement` and `DeleteStatement` that basically implement the `toString()` method to actually build the specific statement. The class diagram of this statements structure is shown in figure 5.9

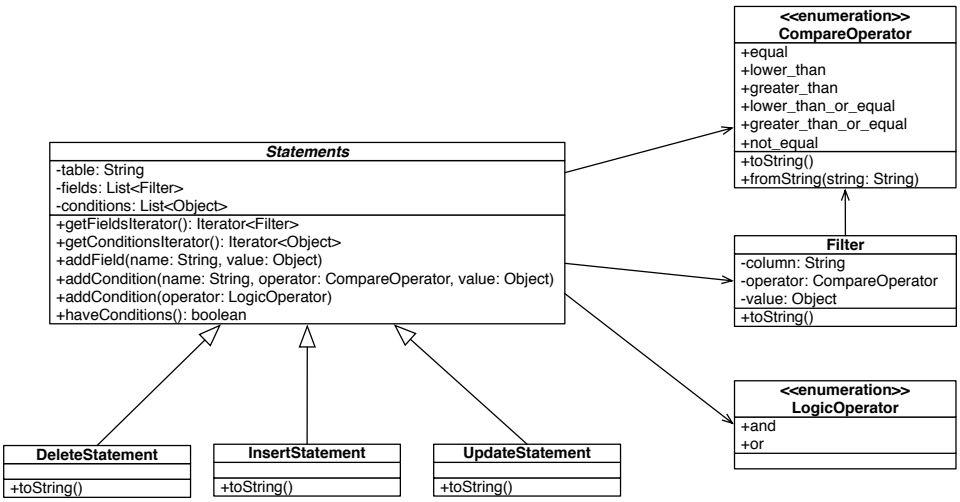


Figure 5.9: Statements structure

The `Statement` class maintains three main fields:

- `table`, that contains the table which the statements refer to;
- `fields`, that maintains a list of elements that represents, in case of an *UPDATE* statements, the the values inside the *SET* clause and, thus, the column names associated with their value; in case of *INSERT* statements, instead, the column names that should be inserted, and their value;

- `conditions`, a linked list of `Filter` elements and `CompareOperator` elements, to represents the *WHERE* clause.

Since not all those elements are needed in all the statements type, specific statements implementations overrides the method that `Statements` provide for handling those fields to deny their usage. For example since the *INSERT* statement does not permit a *WHERE* clause, trying to add a condition on that kind of statements will result in an `UnsupportedOperationException`. Another case is the *DELETE* statements that requires only the *WHERE* clause, so when trying to add a *field* the exception is thrown.

After having defined the statements structure, it is then necessary to provide a way to build the correct instance of a statement, starting by the query or by the operation on an object. To do this in an agile way, a builder class has been implemented.

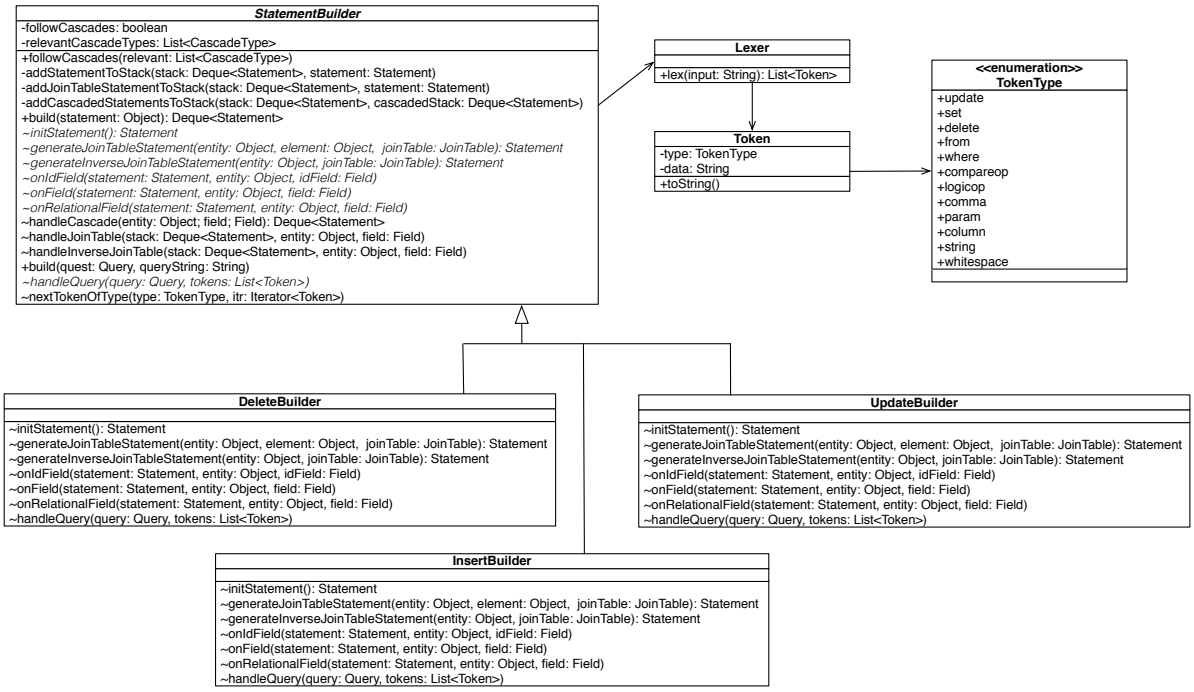


Figure 5.10: Statement builders

The class diagram of the builders, shown in figure 5.10, shows that the same pattern used for statements has been adopted.
The main abstract class `StatementBuilder` provides the facilities to build a generic statement both from object and from a query string. Since many

operations are the same, for all the three types of statements to be built, the `StatementBuilder` class provides an implementation of those common behaviors, and it defines some *abstract* methods that are statement-specific and that needs to be implemented in different ways in the three statement builder classes: `InsertBuilder`, `UpdateBuilder` and `DeleteBuilder`. This degree of abstraction has been possible due to the abstract definition of the `Statement` class which allows to the `StatementBuilder` to act independently from the specific statement type and then to delegate to the specific builder in the cases in which such abstraction is not sufficient anymore.

5.7.1 Build statements from objects

The main issue in generating statements from objects are the cascade types. From the JPA specification [15], the user, on relational fields, can define which type of cascade type he wants to be applied upon operations on the entity. The cascade type can be specified through the annotation `@CascadeType`, four are the relevant values:

- `PERSIST`, when the entity is persisted, every related entity is persisted too, without the need of any explicit persist for that entity;
- `MERGE`, when the entity is updated, every related entity is updated too, without the need of any explicit merge for that entity;
- `REMOVE`, when the entity is deleted, every related entity is deleted too, without the need of any explicit delete for that entity;
- `ALL`, which enclose all the previous types.

The problem in supporting such operations is that statements generated by cascade must keep a logical order, an example is reported in the code snippet 5.4 in which the insert operation for the `Employee` should happen after the insert of the `Department` since the employee maintains a foreign key of the department n which he works.

```
1 INSERT INTO Department (id, name) VALUES ('123', 'Computer_Science')
2
3 INSERT INTO Employee (id, name, department_id) VALUES ('456', 'Fabio', '123')
```

Listing 5.4: Insert statements ordering example

During the development we decided to make the cascade following optional and thus it is configurable in the *migration.xml* file (see the appendix B for further details), but also at run-time, by calling the appropriate methods on the class `BuildersConfiguration`. The statements builders, when created, ask to the `BuildersConfiguration` class in order to decide if the build process should or should not consider the cascade types and, if its the case they set the `relevantCascadeTypes` accordingly. The relevant cascade types have been defined as follow:

1. `ALL` and `PERSIST`, for *INSERT* statements
2. `ALL` and `MERGE`, for *UPDATE* statements
3. `ALL` and `REMOVE`, for *DELETE* statements

The statements execution ordering is the same that should be respected for SQL databases in case foreign key constraints are applied, so, for example, *join table statements* must be taken with particular care since inserts in the join table must happen **after** the insert of the entity itself and deletes in the join table must happen **before** the delete of the entity itself. An example is reported in the code snippet 5.5.

```
1 -- insert entities
2 INSERT INTO Employee (employee_id, name) VALUES ('123', 'Fabio')
3 INSERT INTO Project (project_id, name) VALUES ('456', 'Apollo')
4 -- insert record in the join table
5 INSERT INTO Emp_Proj (id, employee_id, project_id) VALUES ('a', '123', '456')
6
7 -- deleting the employee
8 DELETE FROM Employee_Project WHERE id = 'a'
9 DELETE FROM Employee WHERE employee_id = '123'
```

Listing 5.5: Join table statements ordering example

The abstract builder class `StatementBuilder` provides a single entry point for statements building, by means of the method `build(Object entity)`. This method is designed following a template pattern, the designed general algorithm (reported in algorithm 1) performs all the operations needed to build the statement, and it calls several methods defined as abstract, which are implemented in the specific builders (since they require specific logic), and

several hook methods that can be overrided by the specific builders to change the algorithm behavior.

Algorithm 1 Template algorithm for statements building

```
1: function BUILD(object)
2:     stack ← empty queue
3:     cascadedStack ← empty queue
4:     statemet ← INITSTATEMENT
5:     SETTABLENAME(statement, object)
6:     for all field ← GETFIELDS(statment) do
7:         if ISRELATIONAL(field) & OWNRELATION(field) then
8:             if handle_cascades then
9:                 cascadedStack ← HANDLECASCADE(object, field)
10:            end if
11:            if ISMANYTOMANY(field) then
12:                HANDLEJOINTABLE(stack, entity, field)
13:            else
14:                ONRELATIONALFIELD(statement, entity, field)
15:            end if
16:        else
17:            if ISMANYTOMANY(field) then
18:                HANDLEINVERSEJOINTABLE(stack, entity, field)
19:            end if
20:        end if
21:        if ISID(field) then
22:            ONIDFIELD(statement, entity, field)
23:        else
24:            ONFIELD(statement, entity, field)
25:        end if
26:    end for
27:    ADDSTATEMENTTOSTACK(stack, statement)
28:    if ISID(field) then
29:        ADDCASCADEDSTATEMENTSTOSTACK(stack, cascadedStack)
30:    end if
31: end function
```

Cascade generation is handled through the method `handleCascade(Object entity, Field field)` where `field` is the field of the entity that represents the related entity. The `handleCascade` method checks whenever a statements needs to be generated, based on `relevantCascadeTypes`, and then it recursively calls `build(Object entity)` passing as `entity` the related object.

5.7.2 Build statements from JPQL queries

The main problem for JPQL queries is parsing. Since JPQL is an object querying language, it makes use of an object identifier on which it uses the dot notation to specify the object properties, furthermore JPQL allows the user

to define parameter placeholders (the ones starting with ":") that are filled later through the method `setParameter(String name, Object value)` of the `Query` class.
The translation that should be performed is shown in the code snippet 5.6.

```
-- JPQL query string
UPDATE Test t SET t.name = :name WHERE t.salary >= :salary

-- SQL version
UPDATE Test SET name = 'Fabio' WHERE salary >= '42'
```

Listing 5.6: JPQL to SQL translation

The mapping among *parameter name* and its *value* is kept in the `CloudQuery` and `TypedCloudQuery` classes by intercepting the `setParameter` method on the query.

To solve the parsing problem, the Kundera code was inspected to understand how JPQL queries are parsed, but it turned out they used a custom quite-complex parser, especially for validation purposes. Even looking online no specific JPQL parser has been found so, since we are not interested in validating queries or build complex logic on them, a simple and less time consuming solution was to write a lexer that through regular expressions tokenizes the JPQL string.
Even the building of statements from queries follows a template pattern, the `StatementBuilder` class provides the template method `build(Query query, String queryString)` that tokenizes the query using the lexer and then calls the abstract method `handleQuery(query, tokens)` that is implemented in `DeleteBuilder` and in `UpdateBuilder` which are responsible of iterating over the tokes to build the correct `Statement` instance. The `build(Query query, String queryString)` calls other various hook methods that can be overrided by the specific builders to change the algorithm behavior.

5.7.3 Sending statements to Hegira

Both the method `propagate(Query query)` and `propagate(Object entity, OperationType operation)` fill a statement stack which contains the generated statements in the execution order.

When iterating over the statement stack, statements are extracted one at a time from the head of the stack (LIFO order). Each of the extracted statement is then sent to `Hegira`.

5.8 Interoperability of stored data

The Kundera client described in chapter 4 was developed to be as consistent as possible with the other clients, developed by the Kundera staff, to be more likely accepted by the community and so are not thought to store data to be interpreted also by other clients, and thus are not interoperable. In an optic of data migration, what we want to achieve is that, data stored within a database, and migrated to another one, are still readable from the application without any changes, besides the new database configuration.
The problem for Kundera clients are the relationships. Each database has its own way to define identifier for the persisted entities; for example, in Google Datastore there is the `Key` with a *Kind* and an *identifier*; for Azure there are the *partition-key* and the *row-key*. These concepts are different, but actually quite similar since both databases are key-value columnar databases. A solution to the problem would have been to modify the migration system in order to make it aware of the problem and let it translate the relational columns in the format of the target database; in this way, the relational columns should have been identified in some way to let the system recognize them by adding a pre-defined prefix or a suffix to those columns. Since this solution requires a good amount of changes in the migration system, other solutions have been explored.

Back to the concept of identifier, generally, in columnar databases, columns are grouped in a *column family* and set of columns are identified by a *key*, actually the *key* can span among different *column families*, but that is not the case either in Datastore or Azure Tables. The pair <*column family*, *key*> is sufficient to identify an entity (composed by one or more columns), so a mapping is needed between database-specific terminology to the more general one, this mapping is shown in table 5.1.
At this point we needed a common way of persisting relationships as column family, and key in a way that is interoperable among both the client extension.

General concept	Datastore	Azure Table
Column Family	Kind	partition-key
Key	key-identifier	row-key

Table 5.1: Column family and Key mapping among supported databases.

The proposed solution is to persist a string in the form`columnFamily_key`. This solution has to be preferred with respect to the one that required modification of the migration system since the interoperability is achieved transparently by it.

5.8.1 Kundera clients modification

Since lot of work has already been done on the Kundera clients, the modification to them has been made on a separate branch of the projects named *migration*.

Google Datastore The Datastore extension has been modified to persist relationships as `kind_key-identifier` instead of the `Key` instance. Join tables require particular care; Kundera does not provide the class of the entities involved in the join table, but only the column names and the identifier (the one with the `@Id` annotation). Queries are possible even if the *Kind* is unknown, since Kundera provides the entity class with the entity identifier as arguments to the find operation. To be more consistent, and to apply the newly defined identifier pattern (`kind_key-identifier`) even for join tables, a map is maintained in the client and built inspecting Kundera meta-data, to keep track of which entity classes are involved in which many to many relationship.

Azure Tables The Azure Tables extension has been modified too to reflect the newly defined standard for relationships. In Azure Tables relationships were already being saved as `partition-key_row-key` due to the lack of a class similar to `Key` for Datastore that encapsulate them. The actual problem here is that user can manually handle the partition-key, but it is not a possibility that can be guaranteed anymore since, if an entity is persisted as

`partition-key_row-key`, the *partition-key*, if read by the Datastore extension, will be interpreted as a *Kind*. Since the *Kind* in Datastore extension is the entity table name, it has been decided to lock the Azure Table *partition-key* to the table name, so that the user cannot decide on its own, since this will break the interoperability of data.
The same discussion for the join tables previously made for Datastore , also applies for the Azure Table extension.

5.9 Summary

In this chapter has been rapidly described the CPIM structure and the architecture of the NoSQL service before this work. Then it has been described how it was possible to integrate Kundera as unique persistence provider in the NoSQL service together wit the problems encountered in the process.
From section 5.4 has been described the general interaction we wanted to build to make CPIM and Hegira communicate. Then the architecture and the design choices operated in order to develop such interaction, were introduced and described.

Chapter 6

Evaluation

6.1 Introduction

In this chapter, in section 6.2, will be discussed the tests used to develop the two Kundera extensions. In section 6.3, will be described the YCSB framework that we have used to test the performance of the developed Kundera extensions with respect to the low level API. Finally in section 6.4 we present the application developed to test the data synchronization capabilities of CPIM, while persisting data through the Datastore Kundera extension.

6.2 Test CRUD operations

The Kundera extensions development, due to the lack of information both in the documentation and from the community, has been approached in a test driven way. The first step was to write the required JUnit tests, one for each feature we have planned to support.

We primarily want to achieve code portability of the application model classes, this should be exploited by the usage of the JPA interface but, as stated in chapter 3, there were problems in the old NoSQL service implementation relatively to this point. Secondary we want to be sure that while entities are persisted in the underlying NoSQL database, they can be restored without any loss of information and thus, the mapping between entities and the NoSQL database data model, behave correctly in both verses. Hence the extensions

cannot be tested directly by testing single methods behavior of the extensions classes, this will for sure test the correctness of the operations but, since Kundera clients are not obliged to follow a rigid structure for their code in the implementation of the required interfaces, tests written for a client are not guaranteed to run correctly for another one.
The approach we adopted was to define a single test suite, that will test each one of the feature we planned to support, by interacting directly with Kundera through the JPA interface. This make us able to use the same tests independently of the specific extension and, thus, testing the correctness of CRUD operations through the JPA interface and the portability of the code by means of tests portability.
Those tests have been primarily used to test the extensions during the development phase but, they have also been executed on the remote database instances by connecting to them through the network from the development machine. This tests has been made to guarantee the correct functioning of the two extensions on the real database instances since tests executed locally, are executed against emulators of real systems.

6.2.1 Tests structure

Tests are composed by the entities to be persisted, annotated with the JPA annotations, and a test class for each feature. There are 20 defined entities that includes:

- simple entities related with the JPA relationships annotations, used to test relationships among entities;
- embeddable entities and specific entities that uses those embeddable entities as data types, both used to test the embedded feature of the JPA;
- entities with enum fields, used to test the enum fields support;
- entities declared with different data types for the primary key identifier, used to test ids auto-generation and user-defined ids validation.

The test classes, developed for testing the correctness of relationships, are:

- `MTMTest`, to test the *Many to Many* relationship type;

- `MTOTest`, to test the *Many to One* relationship type;
- `OTMTest`, to test the *One to Many* relationship type;
- `OTOTest`, to test the *One to One* relationship type;

All of those test classes implements two different methods: `testCRUD()`, that test the relationship by interacting with the method of the `EntityManager` interface, and `testQuery()`, that test the relationships by reading, updating and deleting entities through JPQL queries.

The remaining tests classes are:

- `ElementCollectionTest`, that tests the JPA feature of persisting list of objects as entity fields;
- `EmbeddedTest`, that tests the JPA feature of persisting user-defined as entity fields;
- `EnumeratedTest`, that tests the Jsupport for enum fields;
- `QueryTest`, that tests the execution of *SELECT* queries and the support for the various JPQL clauses in queries.

6.3 Performance tests

We wanted to test the overhead of the developed Kundera extensions with respect to direct use of low-level API. To test those kind of performance in terms of throughput and latency of the read and write operations, we have used Yahoo Cloud Serving Benchmark. We choose this approach since it was already used by Kundera developers to estimate the overhead that Kundera adds to the low-level API versions of its clients.

6.3.1 Yahoo Cloud Serving Benchmark

Yahoo Cloud Serving Benchmark (YCSB) is a framework with the general goal of facilitating performance comparisons of the new generations of cloud serving systems [19].

YCSB provides the facility to benchmark various NoSQL database systems such Cassandra, DynamoDB, Voldemort, MongoDB and many others. The key feature of the benchmark system is extensibility, it in facts supports easy definition of new *workloads* and new systems to benchmark.

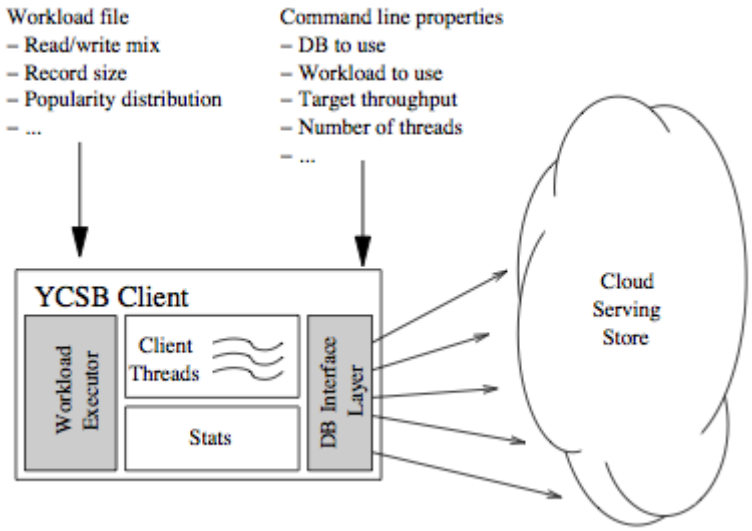


Figure 6.1: YCSB architecture [19]

The access point to the benchmark framework is the *YCSB Client* which is responsible of generating the operations which make up the workload; the workload is then executed by the *Workload executor* that drives multiple client threads which in turn execute a sequential series of operations by making calls to the *database interface layer*.
The workloads are executed in two separate phases:

1. the **load phase**, which loads the workload data to the database system;
2. the **transaction phase**, which execute the workload on the loaded data.

Each thread rules the rate at which it generate requests ad measure the latency and the throughput of its operation. At the end of the benchmark, the statistics module aggregates the measurements and build a report.

6.3.2 YCSB adapters

The YCSB Client abstracts from the specific database system under test through the *database interface layer*; this allows YCSB to generate opera-

tions like “read record” or “update record” without having to understand the specific API of the underlying database.
To create a database adapter the abstract class `com.yahoo.ycsb.DB` must be extended and the following methods needs to be implemented:

- `init()`, which is used to perform any initialization operation such as connecting to the database instance and is called once per thread;
- `read(String table, String key, ...)`, which is supposed to read the given record;
- `scan(String table, String startkey, int recordcount, ...)`, which is supposed to perform a range scan;
- `insert(String table, String key, ...)`, which is supposed to insert the given record;
- `delete(String table, String key)`, which is supposed to delete the given record.

We developed several YCSB adapters:

- an adapter for Google Datastore low-level API
- an adapter for Google Datastore through the developed Kundera extension
- an adapter for Azure Tables low-level API
- an adapter for Azure Tables through the developed Kundera extension

Even if the adapters for Hbase were already been developed by Kundera, they have been both re-written; the Kundera version of the Hbase adapter has been re-written to be identical to the ones for Google Datastore and Azure Tables, the low-level API version, instead, has been re-written because the Kundera client for Hbase has been updated to supported the latest version of the API but the YCSB adapter was not.

Kundera adapters

For the Kundera version of the adapters the same structure has been kept for all three databases. The `EntityManagerFactory` is instantiated at construction, since this operation causes Kundera to initialize all its internal structure; an instance of the entity manager is, instead, created in the `init` method since, the initialization of the `EntityManager`, causes the initialization of the specific Kundera client; in this way each thread will have its own `EntityManager` with which interacting and the same overhead with regards to the database connection operation. Apart from the `scan` method, which were not implemented, every other operations calls the responsible method on the `EntityManager`, in the code 6.1 is shown an example for the insert operation.

```
1  @Override
2  public int insert(String table, String key, ...) {
3      ...
4      try {
5          AzureTableUser user = new AzureTableUser(key, nextString(),, ...);
6          em.persist(user);
7          if (timeToClearEntityManager()) {
8              em.clear();
9          }
10         return OK;
11     } catch (Exception e) {
12         return ERROR;
13     }
14 }
```

Listing 6.1: Insert operation of the Azure Tables adapter

In the code 6.1 there are two elements that needs a deep explanation. The first thing to notice is that is persisted an instance of `AzureTableUser`, in fact, we were not able to persist the entities generated by YCSB because, to be able to persist an entity with the JPA, we need an annotated class which should be then listed in the *persistence.xml*. For this reasons three different user class and three different persistence units has been defined:

- `AzureTableUser`, which refer to `kundera_azure_pu`, the persistence unit with the configuration for Azure Tables;
- `DatastoreUser`, which refer to `kundera_datastore_pu`, the persistence unit with the configuration for Google Datastore;

- `HBaseUser`, which refer to `kundera_hbase_pu`, the persistence unit with the configuration for Hbase.

The second thing to notice is the call to the `timeToClearEntityManager()` method, which returns `true` each time 500 entities are persisted, this causes a clear of the persistence cache by calling `EntityManager.clear()`. If this operation is not performed, entities read can occur within the persistence cache bypassing the request to the underling database. We choose to clear the cache every 500 entities, in our workloads of 100.000 entities, to maintains the same proportion with the one used by Kundera in their test, in which the persistence cache is cleared every 5.000 entities on workloads of 1 million operations.

Low-level API adapters

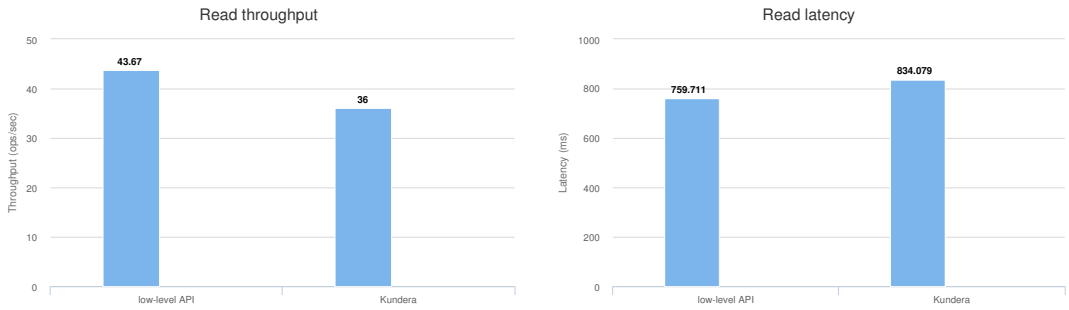
Also the adapters for the low-level API version follows the same general structure. The connection to the database is performed, through low-level API, in the `init` method, to have a common behavior with respect to the Kundera adapters. Read, insert and delete operations are performed by a call to the specific low-level API, while the `scan` method has not been implemented.
The `init` method uses the properties defined in the property files specified in the execution command of the benchmark, to locate the remote database and instantiate a connection.

6.3.3 YCSB tests

YCSB comes with a core set of workloads, each workloads represents a particular mix of read and write operations and define the total number of operations that should be executed. YCSB benchmarks are executed in two separate phases and each of them generates a report. From the report of the *load* phase, since this phase is responsible of storing the data required to run the workload to the target database, we obtains information about the throughput and the latency for the write operation; from the *transaction* phase we want to obtain information about throughput and latency for the read operation. To do this we run a custom workload composed of 100.000 operations entirely of type read so that the *transaction* phase will generates the statistics we need. Defined the adapters and the workload we were able to execute the tests.

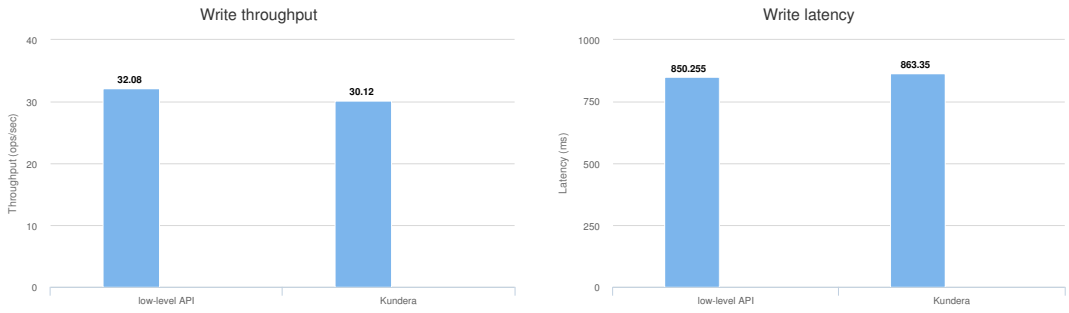
Google Datastore tests

The tests for Google Datastore has been executed over a remote Datastore instance in an application billed by Politecnico di Milano and configured to accept remote API execution.
The results of the tests are reported in figure 6.2 for the read operation and in figure 6.3 for the write operation.



(a) Throughput (b) Latency

Figure 6.2: Google Datastore - read operation benchmark results

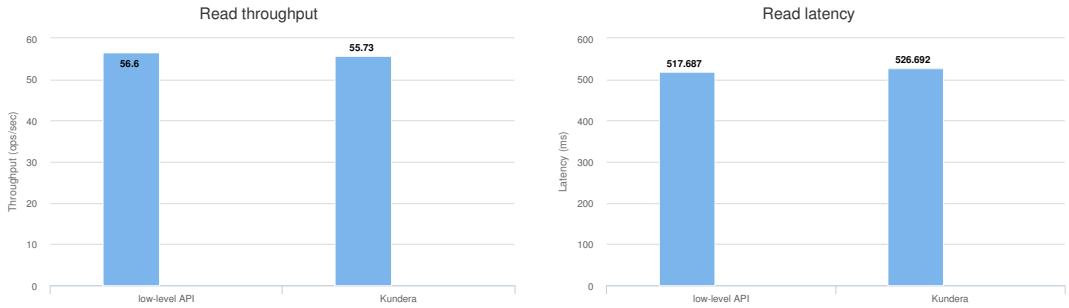


(a) Throughput (b) Latency

Figure 6.3: Google Datastore - write operation benchmark results

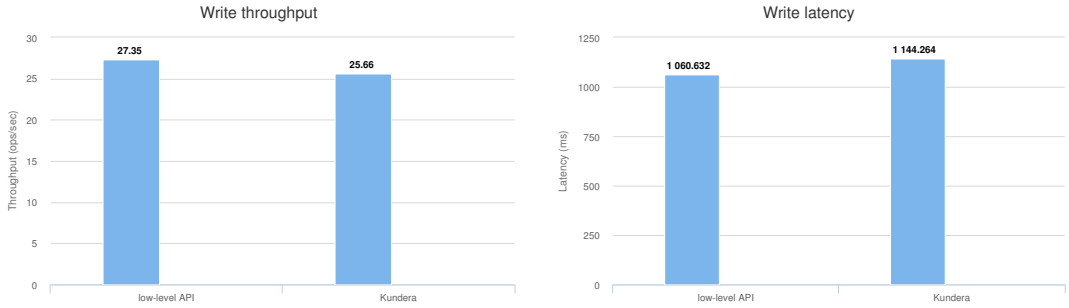
Azure Tables tests

The tests for Azure Tables has been executed over a remote storage instance deployed in Azure from the billing account of Politecnico di Milano.
The results of the tests are reported in figure 6.4 for the read operation and in figure 6.5 for the write operation.



(a) Throughput (b) Latency

Figure 6.4: Azure Tables - read operation benchmark results



(a) Throughput (b) Latency

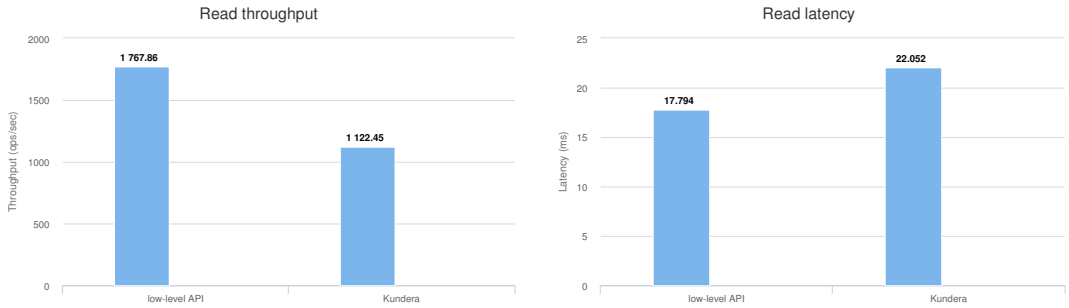
Figure 6.5: Azure Tables - write operation benchmark results

Hbase tests

Hbase test should have been executed over an instance of Hbase, in full distributed configuration, in the cloud of Politecnico di Milano but due to a failure of the host machines the tests cannot be performed.
In figure 6.6 and 6.7 are reported the results obtained while testing the Hbase adapters. They have been executed on a workload of 1.000 entities in a locally installed instance of Hbase.

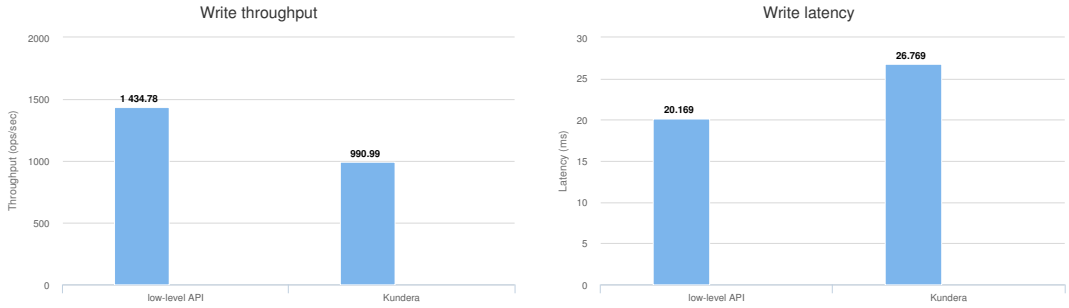
6.3.4 Discussion

The main objective of our tests was to guarantee that the loss of performance, between the Kundera version of the client and the client written with direct



(a) Throughput (b) Latency

Figure 6.6: Hbase - read operation benchmark results



(a) Throughput (b) Latency

Figure 6.7: Hbase - write operation benchmark results

use of the low-level API, was minimum or at least not as much to discourage the use of the JPA approach for NoSQL databases.

Since Kundera provides a set of benchmark results obtained through YCSB and due to the fact that they shows an acceptable performance loss, our objective included the comparison of our results with the results obtained by Kundera. Kundera executes its tests on instances of the databases that reside on the testing machine, since is meaningless for us to test the developed extensions against their local emulator, we cannot directly compare our results with the Kundera ones.

Thus we decided to replicate the results of Kundera on Hbase, executing the tests, no more on a local instance of Hbase but on a remote instance that was deployed in the Politecnico di Milano cloud. This will make us able to compare our tests executed over remote instances of the NoSQL database with

the replicated version of tests on Hbase.
Unfortunately this comparison cannot be done due to two problems that raised during the development. The first problem we encountered was relative to the version of the client that were tested by Kundera, the tests were executed with the version 2.6 of Kundera but, as we write, Kundera have reached version 2.16. This caused an incompatibility between the code of the tests and the low-level API of Hbase and thus we needed to re-write the low-level API YCSB adapter for Hbase. Finally we faced a problem on the Hbase instance on the Politecnico di Milano cloud that would have required a complete re-install of the Hbase cluster.

As can be seen comparing the values of throughput of Hbase and of the other tests, while tests are executed over the network, throughput drops significantly. YCSB tries to go as fast as possible in issuing operation on the database instance trying to reach the maximum throughput the system can afford. The cause of the throughput drop is due to the high number of TCP connections that needs to be maintained during the benchmark execution, in fact, each requests needs a TCP connection to be opened and maintained at leas until the response comes from the server. When executing the tests with 100.000 entities this becomes the bottleneck for the benchmark.
Similar considerations holds for the latency, the values reported for Google Datastore and Azure Tables, includes the round trip time of the requests and furthermore it depends on the state of network congestion. By looking at the network consumption data reported by the operating system of the testing machine, we estimated that the average round trip time, during the tests, was of 56,29 ms.

After these considerations and by looking at the differences in both throughput and latency for the low-level API and the developed Kundera extension we conclude that a performance loss exists, but is not enough to discourage the usage of those NoSQL databases through Kundera, especially given the benefits this solution bring in working with NoSQL databases.

6.4 Hegira generator

To be able to test the interaction between the CPIM library and the synchronization system, and to provide an example of usage of the extended CPIM library, we have developed *Hegira generator*.
The application provides two behaviors through command line interface, a `clean` command, to clean-up the remote Datastore instance by deleting all the entities of all Kinds, and a `generate` command, that take as argument the number of entities to be generated per table and generates them.

Data generation is done upon a pre-defined entity model inside the application and described by the ER diagram of figure 6.8. Build an entity generator agnostic to the entities model was not our goal and it would have required a way to automatically build the dependency graph of the entities since entities related to other ones, should have a reference to the entity they depends on. Hence the application is aware of the entities dependencies and generates them accordingly.

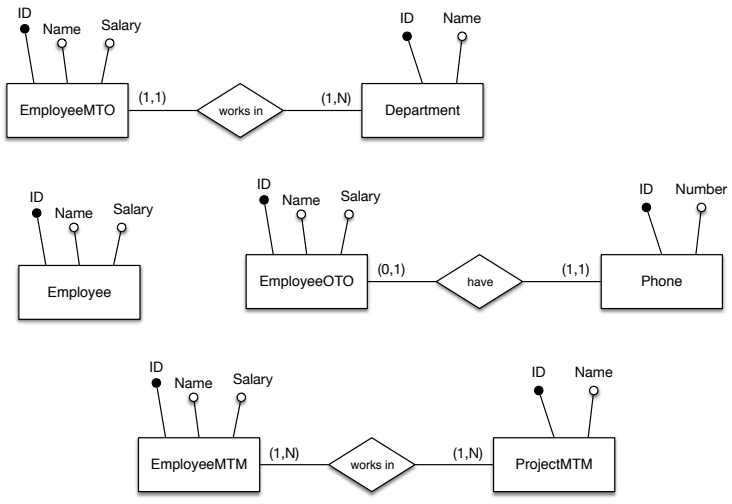


Figure 6.8: ER diagram of Hegira-generator model

To be able to generate random entities, two methods are used:

- `persist(Class master)` that generates and persist entities without dependencies (such as `Employee` in the application model);

- `persist(Class master, Class slave, DependencyType type)` that generates and persist the entities of the *master* class and then uses randomly extracted entities among those just generated to fill the dependencies for the entities of the *slave* class. The `DependencyType` would be `SINGLE`, if the *slave* class needs just one element to fill the dependency (which is the case of *One to One* and *Many to One* relationships), or `COLLECTION`, if the *slave* class needs more than one element to fill the dependency (which is the case for *Many to Many* relationships).

The actual entity generation is delegated to the entity itself through reflection since each entity of the model implements the `Randomizable` interface. An example of entity genearion through this interface is shown in the snippet 6.2.

```
1  @Entity
2  public class EmployeeOTO implements Randomizable<EmployeeOTO, Phone> {
3      ...
4      @Override
5      public EmployeeOTO randomize(Phone dependency) {
6          setName(RandomUtils.randomString());
7          setSalary(RandomUtils.randomLong());
8          setPhone(dependency);
9          return this;
10     }
11 }
```

Listing 6.2: Entities generation

6.4.1 Exploited CPIM features

To perform the persist operation of the generated entities is used the `EntityManager` interface on which is called the `persist` method. This is completely JPA compliant and the user is not aware of what is done under the hood since communication with the synchronization system is handled automatically. An example is provided in the code 6.3.

```
1 CloudEntityManager em = MF.getFactory().getEntityManager();
2 Department dep = new Department("Computer Science")
3 em.persist(dep)
```

Listing 6.3: Persisting entities in CPIM

The persist operation through `CloudEntityManager` contacts the synchronization system to get the assigned sequence numbers for the specific tables and assign the first of them to the entity before delegating to Kundera the persist operation.

The application make use of the possibility of modifying at run-time the size of sequence numbers range that is requested to the synchronization system. Hence before the persist operation, the size of the sequence number range is set to the double of the number of entities to be generated. This is done through a call to `SeqNumberProvider.getInstance().setOffset(tableName, offset)`, if the resulting range size is grater than the maximum size that can be requested, is limited to that value.

The last feature that is exploited by the application is the backup to file of the sequence numbers. The backup is configured in the `migration.xml` file, as described in the appendix B. This permit to the application, when is restarted, to restore the sequence numbers without the need of contacting the synchronization system. Furthermore, to avoid execution of the persist operations on a table which entities generation was completed in a previous execution, a file with the list of the table completely generated is kept in the same folder specified for the sequence numbers backup files.

6.5 Summary

In this chapter we have presented the test driven approach used to develop the Kundera extensions. Then has been described how we prepared and executed the test of correctness and performance made for the two developed Kundera extension showing the minimal performance loss that Kundera add to the low-level API. Finally we have presented *Hegira-generator*, the application developed to test the mechanisms encapsulated inside CPIM to interacts with the synchronization system.

Chapter 7

Conclusions and future Works

This work presented an approach that allow the users of the CPIM library to interact with different NoSQL solution through a common interface, identified in the JPA interface. This was possible by exploiting the functionalities of Kundera, a JPA compliant ORM built for NoSQL databases.
In the state of the art analysis of chapter 2, NoSQL database has been presented as an alternative to RBDMS and, the necessity of a common interface to communicate with different NoSQL solutions has been highlighted, while presenting the different solutions available in this direction.
Chapter 3 provide a detailed description of the motivation that lead to the necessity of modifying the NoSQL service of the CPIM library and to the decision of integrating the migration and synchronization system *Hegira* as part of the NoSQL service.
In Chapter 4 has been described what Kundera is, its architecture and, has been descried in detail, the development process that lead to the two new Kundera extensions, the first one to support Google Datastore and the second one to support Azure Tables. The extensions has been developed as part of a more general work on the CPIM library, indeed the develop of those Kundera extension was aimed to maintain the CPIM support for Google Datastore and Azure Tables. Chapter 5 described in detail the work made on the NoSQL service of CPIM, which is about the integration of Kundera as the unique persistence provider, and the integration of *Hegira* to support transparent data synchronization and migration.
Finally, chapter 6 shows the performance tests executed over the developed Kundera extensions with respect to the use of the low-level API, using the

YCSB framework. The results showed that the overhead introduced by Kundera, and by the client extension, in terms of operation throughput and latency, is absolutely acceptable to justify the benefit that the use of Kundera introduces. Furthermore, the chapter described *Hegira-generator*, the application developed to generate data and testing the interaction with CPIM and *Hegira*.

Possible future works should continue on both CPIM and Kundera. Indeed, CPIM needs to be updated to interacts with the latest version of the various cloud provider API and some components needs to be rewritten, as explained in section 5.3.1 for the Queue service.
Further work can also be done in intercepting the user queries, that are then sent to the migration system, supporting for example the *criteria API* discussed in section 5.5.2.
Finally some work can be done in adding to Kundera the support for more NoSQL databases such as Dynamo DB.

Appendices

Appendix A

Configuring Kundera extensions

A.1 Introduction

In this appendix are described in detail the configurations available for the two developed Kundera extensions. Are described the required properties that needs to be configured in the *persistence.xml* file and the available properties that can be defined in the external datastore specific properties file.

A.2 Common configuration

The main configuration is performed in the *persistence.xml* file and it follows the JPA standard. The template of the file is as follow:

```
1  <?xml version="1.0" encoding="UTF-8" standalone="no"?>
2  <persistence ... >
3      <persistence-unit name="...">
4      <provider>com.impetus.kundera.KunderaPersistence</provider>
5          <class> ... </class>
6          <exclude-unlisted-classes>true</exclude-unlisted-classes>
7          <properties>
8              <!-- kundera properties -->
9          </properties>
10     </persistence-unit>
11 </persistence>
```

Listing A.1: persistence.xml template

A name for the persistence unit is mandatory as it will be referenced inside the classes of the model as shown in the snippet A.2 in which has been declared as schema the `kundera.keyspace` property to the persistence unit name. The full

name of the classes that needs to be handled through this persistence unit must be specified in the `<class>` tag. Each extension needs different configuration that needs to be specified inside the `<properties>` tag.

```
1  @Table(schema = "gae-test@pu")
2  public class Employee {
3
4      @Id
5      @Column(name = "EMPLOYEE_ID")
6      private String id;
7
8      @Column(name = "NAME")
9      private String name;
10
11     @Column(name = "SALARY")
12     private Long salary;
13 }
```

Listing A.2: Declaring the schema

A.3 GAE Datastore

Two configuration are possible:

1. use the datastore instance within the app engine application;
2. use a remote datastore instance through remote API.

The properties to be specified inside the `<properties>` tag for the first case are:

- `kundera.client.lookup.class` (*required*), must be set to `it.polimi.kundera.client.datastore.DatastoreClientFactory`;
- `kundera.ddl.auto.prepare` (*optional*), possible values are:
  - `create`, which creates the schema (if not already exists);
  - `create-drop`, which drop the schema (if exists) and creates it;
- `kundera.client.property` (*optional*), the name of the xml file containing the datastore specific properties.

In addition to the previous properties and in case of remote API, those properties are also necessary:

- `kundera.nodes` (*required*), url of the app engine application on which the datastore is located;
- `kundera.port` (*optional*) default is **443**, port used to connect to datastore;
- `kundera.username` (*required*), username of an admin on the remote server;
- `kundera.password` (*required*), password of an admin on the remote server.

To test against local app engine run-time emulator the configuration is as follow:

```
1 <property name="kundera.nodes" value="localhost"/>
2 <property name="kundera.port" value="8888"/>
3 <property name="kundera.username" value="username"/>
4 <property name="kundera.password" value=""/>
```

Listing A.3: GAE Datastore emulator configuration

in this case the value for `kundera.password` does not matter.

Datastore specific properties file

A file with client specific properties can be created and placed inside the classpath, its name must be specified in the *persistence.xml* file through the property `<property name="kundera.client.property" value="filename.xml"/>`. The template of the file is the following:

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <clientProperties>
3      <datastores>
4          <dataStore>
5              <name>datastore</name>
6              <connection>
7                  <properties>
8                      <property name="..." value="..."></property>
9                  </properties>
10             </connection>
```

```
11          </dataStore>
12      </datastores>
13  </clientProperties>
```

Listing A.4: GAE Datastore - datastore specific configuration

The available properties are:

- `datastore.policy.read` (`optional`) [eventual|strong] default is **strong**. Set the read policy;
- `datastore.deadline` (*optional*). RPCs deadline in seconds;
- `datastore.policy.transaction` (*optional*) [auto|none] default is **none**. Define if use implicit transaction.

A.4 Azure Table

The properties to be specified inside the `<properties>` tag are:

- `kundera.username` (*required*), the storage account name available from azure portal;
- `kundera.password` (*required*), the storage account key available from azure portal;
- `kundera.client.lookup.class` (*required*), must be set to `it.polimi.kundera.client.azuretable.AzureTableClientFactory`;
- `kundera.ddl.auto.prepare` (*optional*), possible values are:
  - `create`, which creates the schema (if not already exists);
  - `create-drop`, which drop the schema (if exists) and creates it.
- `kundera.client.property` (*optional*), the name of the xml file containing the datastore specific properties.

Datastore specific properties file

A file with client specific properties can be created and placed inside the classpath, its name must be specified in the *persistence.xml* file. The template of the file is the following:

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <clientProperties>
3      <datastores>
4          <dataStore>
5              <name>azure-table</name>
6              <connection>
7                  <properties>
8                      <property name="..." value="..."></property>
9                  </properties>
10             </connection>
11         </dataStore>
12     </datastores>
13 </clientProperties>
```

Listing A.5: Azure Tables - datastore specific configuration

The available properties are:

- `table.emulator` (*optional*) [true|false] default is **false**. If present (and set to true) storage emulator is used. When using development server `kundera.username` and `kundera.password` in *persistence.xml* are ignored;
- `table.emulator.proxy` (*optional*) default is **localhost**. If storage emulator is used set the value for the emulator proxy;
- `table.protocol` (*optional*) [http|https] default is **https**. Define the protocol to be used within requests;
- `table.partition.default` (*optional*) default is **DEFAULT**. The value for the default partition key, used when no one is specified by the user.

Appendix B

Configuring CPIM migration

B.1 Introduction

In this appendix is presented the new configuration file added to CPIM to support the various configurations available for the interaction with the migration system.

B.2 *migration.xml*

The template of the *migration.xml* file is the following:

```
1  <?xml version="1.0" encoding="UTF-8" standalone="no"?>
2  <migration>
3      <zooKeeper>
4          <type>...</type>
5          <connection>...</connection>
6          <range>...</range>
7      </zooKeeper>
8      <backup>
9          <execute>...</execute>
10         <type>...</type>
11         <directory>...</directory>
12         <prefix>...</prefix>
13     </backup>
14     <followCascades>...</followCascades>
15 </migration>
```

Listing B.1: migration.xml template

Three are the main section that can be configured:

1. ZooKeeper client;
2. sequence number backup;
3. follow cascades while build statements.

The first two options are the most complex and are described in the following sections, the third option can assume be *true* or *false* but is *optional* since is set to **false** by default. In case the value is set to *true*, the statement builders when builds the statements from objects will read the values specified for the `@CascadeType` annotation and, if necessary, builds the cascade statements and sends them to *Hegira* in the correct execution order as described in 5.7.

B.2.1 Configure the ZooKeeper client

For ZooKeeper client, must be chosen the *thread* or the *http* version as described in 5.6.

Thread version An example configuration would be the following one:

```
1 <zooKeeper>
2     <type>thread</type>
3     <connection>localhost:2181</connection>
4     <range>50</range>
5 </zooKeeper>
```

Listing B.2: ZooKeeper - thread type configuration

The `<connection>` tag is **required** and should contains an address in the form `host:port` which should be the host address and the port on which the ZooKeeper service is running.
The `<range>` is *optional* since the default is **10** and is the default dimension of the range that `SeqNumberDispenser`(s) use to ask sequence numbers to the synchronization system.

HTTP version An example configuration would be the following one:

```
1 <zooKeeper>
2     <type>http</type>
3     <connection>http://server.com/hegira-api/zkService</connection>
4     <range>50</range>
5 </zooKeeper>
```

Listing B.3: ZooKeeper - http type configuration

In this case the `<connection>` tag should contains the API base path of the server in which the service is running.
For the `<range>` tag the same consideration made for the thread case applies.

B.2.2 Configure a sequence number backup

A sequence number backup can be configured either on a blob storage (when running on PaaS) or to file (when running on IaaS).

The `<execute>` tag define if backup should or should not be performed, the possible values are *yes* or *no*. This tag is *optional* since its default is **yes**. In case backups must be turned off, the configuration should be the following:

```
1 <backup>
2     <execute>no</execute>
3 </backup>
```

Listing B.4: Turning off sequence numbers backup

For the other configuration options, two cases must be distinguished:

Backup to Blob An example configuration would be the following one:

```
1 <backup>
2     <type>blob</type>
3     <prefix>SeqNumber_</prefix>
4 </backup>
```

Listing B.5: Backup to blob

The text in the `<prefix>` tag will be prefixed to each blob created to avoid file name conflicts. This prefix is *optional* field as default is set to **SeqNumber_**.

Backup to file An example configuration would be the following one:

```
1 <backup>
2     <type>file</type>
3     <directory>/backups</directory>
```

```
4       <prefix>SeqNumber_</prefix>
5   </backup>
```

Listing B.6: Backup to file

The `<directory>` tag is **mandatory** and must contain the path to the directory in which the backup files should be stored.
For the `<prefix>` tag the same considerations made for backup to blob applies.

B.3 Use CPIM without migration system

The *migration.xml* file is not necessary if the user would not use the migration system. If the file is not present inside the `META-INF` folder, the NoSQL service will interact only with the underlying persistence provider without instantiating any of the classes required for the interaction with the migration system. This is possible due to the lazy initialization of those components that are initialized only the first time are actually used since are built with a singleton pattern.

Appendix C

Run YCSB tests

C.1 Introduction

In this appendix is descried the required procedure to build the benchmark project that contains the YCSB adapters, then are shown the required commands to be executed in order to execute the tho phases of an YCSB benchmark and the available values to be configured for each of the supported adapters.

C.2 Preliminary operations

In order to build the benchmark project, (available at `https://github.com/Arci/kundera-benchmark`) some libraries needs to be downloaded since are not available in any maven repository:

- Azure Table extension
  `https://github.com/Arci/kundera-azure-table`
- GAE Datastore extension
  `https://github.com/Arci/kundera-gae-datastore`

The Azure Table extension tests to run requires a reachable storage emulator on Windows so if this is not possible, skip tests by running `mvn clean install -DskipTests`.
Tests for the Datastore extension can be executed without any configuration as they are executed thought Google in-memory Datastore stub.

Also YCSB is not available in any maven repository, it must be downloaded (`https://github.com/brianfrankcooper/YCSB`) and installed locally, always through `mvn install`.

When all the required dependency for kundera-benchmark are resolved, is possible to install it with `mvn clean install` and then lunch the command `mvn dependency:copy-dependencies`, this will create a directory called `dependency` in the `target` directory containing all the jars of the dependencies. The `dependency` folder will be used for defining the classpath later on.

C.3 Run tests for low-level API version

The two phases of the YCSB benchmark can be executed through the command:

```
java -cp KUNDERA_BENCHMARK_JAR_LOCATION:PATH_TO_DEPENDENCY_FOLDER/*
com.yahoo.ycsb.Client -t -db DATABASE_ADAPTER_CLASS_TO_USE
-P PATH_TO_WORKLOAD -P PATH_TO_PROPERTY_FILE
-s -threads THREAD_TO_USE -PHASE > OUTPUT_FILE
```

Listing C.1: Run low-level API benchmarks

where PHASE should be `load` for **load** phase or `t` for **transaction** phase.

Available adapter classes are:

- `it.polimi.ycsb.database.AzureTableClient` for Azure Table;
- `it.polimi.ycsb.database.DatastoreClient` for GAE Datastore;
- `it.polimi.ycsb.database.KunderaHBaseClient` for Hbase.

C.3.1 Property files

As can be seen from the command, a property file must be specified. Properties files must provide information to locate the database to test when running the benchmarks on the low-level API versions.

Google Datastore The available properties are:

- `url` (*required*);
- `port` (*optional*), default is 443;
- `username` (*required*), the username of an admin on the remote application;
- `password` (*required*), can be omitted if tests are against localhost.

Azure Table The available properties are:

- `emulator` (*optional*) [true|false];
- `account.name` (*required*) if not using emulator, available from azure portal;
- `account.key` (*required*) if not using emulator, available from azure portal;
- `protocol` (*optional*) [http|https], default is https.

If `emulator` is set to *true* the remaining properties are ignored.

Hbase The properties must be configured inside the adapter class because, to be more accurate w.r.t. the Kundera client, connection cannot be done in the `init()` method.
The properties can be set modifying the following constants:

- `node`, the master node location;
- `port`, the master node port;
- `zookeeper.node`, the node location for `hbase.zookeeper.quorum`;
- `zookeeper.port`, the node port for `hbase.zookeeper.property.clientPort`.

Since property file is not needed for Hbase, it does not need to be specified while running the benchmarks.

C.4 Run tests for Kundera version

Two phases of the YCSB benchmark can be executed through the command:

```
java -cp KUNDERA_BENCHMARK_JAR_LOCATION:PATH_TO_DEPENDENCY_FOLDER/*
com.yahoo.ycsb.Client -t -db DATABASE_ADAPTER_CLASS_TO_USE
-P PATH_TO_WORKLOAD -s -threads THREAD_TO_USE -PHASE > OUTPUT_FILE
```

Listing C.2: Run Kundera clients benchmarks

where PHASE should be `load` for **load** phase or `t` for **transaction** phase.

Available adapter classes are:

- `it.polimi.ycsb.database.KunderaAzureTableClient` for kundera-azure-table extension;
- `it.polimi.ycsb.database.KunderaDatastoreClient` for kundera-gae-datastore extension;
- `it.polimi.ycsb.database.KunderaHBaseClient` for kundera-hbase extension.

C.4.1 *persistence.xml* configuration

In the *persistence.xml* file each persistence unit must be configured to locate the database to test.
The possible configurations are described in the appendix A.
Hbase configuration make use also of a datastore specific property file `hbase-properties.xml` in which can be configured the value for `hbase.zookeeper.quorum` and `hbase.zookeeper.property.clientPort`.

Bibliography

[1] Azure table storage. http://azure.microsoft.com/en-us/documentation/articles/storage-java-how-to-use-table-storage. [Online].

[2] Google app engine datastore. https://cloud.google.com/datastore/docs/concepts/overview. [Online].

[3] nosql-database.org. http://nosql-database.org/. [Online, Accessed 14 March 2015].

[4] Using jpa with app engine. https://cloud.google.com/appengine/docs/java/datastore/jpa/overview. [Online].

[5] Kundera. https://github.com/impetus-opensource/Kundera, 2010. [Online].

[6] Unql. http://unql.sqlite.org/, 2011. [Online].

[7] Apache gora. http://gora.apache.org/index.html, 2012. [Online].

[8] Playorm. http://buffalosw.com/products/playorm, 2012. [Online].

[9] Apache metamodel. http://metamodel.apache.org/, 2013. [Online].

[10] Spring data. http://projects.spring.io/spring-data/, 2013. [Online].

[11] Apache phoenix. http://phoenix.apache.org/index.html, 2014. [Online].

[12] Elisabetta Di Nitto, Marco Scavuzzo, Michele Ciavotta, Fabio Arcidiacono. Modacloudml package on data partition and replication. Technical report, Politecnico di Milano, September 2014.

[13] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, Robert E. Gruber. Bigtable: A distributed storage system for structured data. Technical report, Google, 2006.

[14] Filippo Giove, Davide Longoni. Un approccio per lo sviluppo di applicazioni portabili per sistemi di cloud computing. Master's thesis, Politecnico di Milano, 2012.

[15] Keith Mike, Schincariol Merrick. *Pro JPA 2*. Apress, Berkely, CA, USA, 2nd edition, 2013.

[16] Paolo Atzeni, Francesca Bugiotti, Luca Rossi. Uniform access to non-relational database systems: the sos platform. Technical report, Università Roma Tre, 2012.

[17] Marco Scavuzzo. Interoperable data migration between nosql columnar databases. Master's thesis, Politecnico di Milano, 2013.

[18] Jeff Schnitzer. Objectify. `https://code.google.com/p/objectify-appengine/wiki/IntroductionToObjectify`. [Online].

[19] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, Russell Sears. Benchmarking cloud serving systems with ycsb. Technical report, Yahoo! Research, 2010.

[20] Hitachi Data Systems. Reduce costs and risks for data migrations. Technical report, Hitachi Data Systems, March 2014.