Politecnico di Milano

Facoltà di Ingegneria dell'Informazione

Corso di Laurea Magistrale in Ingegneria Informatica
Dipartimento di Elettronica, Informazione e Bioingegneria

. . . Titolo della tesi . . .
. . . al massimo su due righe . . .

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, 1 Aprile 2005

Fabio.

Estratto

abstract in italian

Abstract

abstract in english

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Introduzione al lavoro. Inizia direttamente, senza nessuna sezione.
Argomenti trattati suddivisi sezione per sezione...

Original Contributions

This work include the following original contributions:

- ... riassunto sintetico dei diversi contributi
- ...
- ...

Outline of the Thesis

This thesis is organized as follows:

- In Chapter 2 is described the current evolution of NoSQL databases . 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 SQL-99 for SQL databases.
- In Chapter 3 ...
- In Chapter 4 is dedicated to the develop of the two Kundera client extension that have been developed in order to support Google Datastore

and Azure Tables that will be then used in CPIM as adapters for the relative database.

- In Chapter 5 is presented the work made on CPIM. As a first step is described a modification in the CPIM NoSQL service aimed to integrate Kundera as unique persistence layer for NoSQL access using the standard JPA interface, the library was previously using several different JPA implementation one for each of the supported databases. Furthermore is discussed the extension of CPIM to include an interaction with the migration system *Hegira*.
- In Chapter 6 ...
- 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 and compared with SQL solutions. In section 2.3 are listed some of the solution that has been developed in defining a common language or interface to interact with different NoSQL databases. Finally the CPIM library is introduced as a tentative in defining a common interface to interacts with different vendor in a PaaS environment which include accessing the different NoSQL solution of the PaaS provider.

2.2 NoSQL databases

2.2.1 NoSQL motivations

2.2.2 NoSQL characteristics

2.2.3 Standard language

ORM (Object Relational Mapping) solutions came into existence to solve OO-impedance mismatching problem. Most popular among them are Hibernate, Toplink, EclipseLink etc. They worked beautifully with relational databases like Oracle and MySQL, among others.

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.

Problem with NoSQL databases is that there is NOT EVEN ONE existing industry standard (like SQL) for them. The very basic idea of something opposed to SQLand as a result deviation from standards and rules, is going to be suicidal, if not corrected at right time. Learning to work with a new NoSQL database is always cumbersome as a result.

Apart from that, people lack in-depth knowledge of NoSQL. Even if they do, they are confined to one or two. In relational world, people depend upon their knowledge of SQL and JDBC to work on basic and intermediate database things. Switching to another database requires little or almost no effort, which otherwise is painful in NoSQL world.

ORM for NoSQL is a bit mis-leading term. People prefer to call it OM tool for NoSQL or maybe ODM Object data-store Mapping tool. ORM frameworks have already been there for 30+ years and its a de-facto industry standard. People are very clear about what ORM tools are supposed to do. There are no surprises.

Key here is to let people forget worrying about complexities inherent in NoSQLs. Let them do things in a way they already know and are comfortable with. Why not use an approach that is there for this problem domain for decades and has proven its usefulness.

A good use case advocating use of ORM tools is migration of applications (built using ORM tool) from RDBMS to NoSQL database. (or even from one NoSQL database to another). This requires (at least in theory) little or no programming effort in business domain.

2.3 Approaches for a common language

The lack of a common language and of a standardization as SQL-99 is for SQL has bring developers to build many different solutions with slightly different approaches. Many of the solutions that will be discussed are open source projects developed and maintained by a community, some others are approaches that came from academic researches and some other are commercial solution.

Apache Phoenix is a SQL query engine for accessing NoSQL datastores such as Apache HBase. It is accessed as a JDBC driver and enables querying,

updating, and managing NoSQL tables through standard SQL. Instead of using map-reduce, Apache Phoenix compiles your SQL query into a series of HBase scans and orchestrates the running of those scans to produce regular JDBC result sets. Direct use of the HBase API, along with coprocessors and custom filters, results in performance on the order of milliseconds for small queries, or seconds for tens of millions of rows.

Apache Phoenix is a relational database layer over HBase delivered as a client-embedded JDBC driver targeting low latency queries over HBase data. Apache Phoenix takes your SQL query, compiles it into a series of HBase scans, and orchestrates the running of those scans to produce regular JDBC result sets. The table metadata is stored in an HBase table and versioned, such that snapshot queries over prior versions will automatically use the correct schema. Direct use of the HBase API, along with coprocessors and custom filters, results in performance on the order of milliseconds for small queries, or seconds for tens of millions of rows.

CQL: Cassandra Query Language (CQL) is a query language for the Cassandra database.

GQL: GQL is a SQL-like language for retrieving entities or keys from Datastore. While GQL's features are different from those of a query language for a traditional relational database, the GQL grammar is similar to that of SQL.

—¿ tendenza ad usare SQL come linguaggio comune

2.3.1 Kundera

Kundera is a JPA 2.1 compliant object-datastore mapping library for NoSQL datastores. Kundera makes working with NoSQL databases simple and fun. Kundera does not reinvent the wheel by making another client library; rather it leverages the existing libraries, and builds on top of them a wrap-around API to developers do away with the unnecessary boiler plate codes, and program a neater, cleaner code that reduces code-complexity and improves quality. And above all, improves productivity.

Kundera supports cross-datastore persistence. This means you can store and fetch related entities in different datastores using a single method call.

Kundera is JPA 2.1 compatible. It strictly uses JPA annotations to map your objects into your datastore tables(Did I say table? Huh! looks like a

relational database term. well, we prefer this as a general name since different NoSQL datastores use different naming - Column family for Cassandra, Table for HBase and collections for MongoDB)

2.3.2 Spring-data

Makes it easy to use new data access technologies, such as non-relational databases, map-reduce frameworks, and cloud based data services. Spring Data also provides improved support for relational database technologies. This is an umbrella project which contains many subprojects that are specific to a given database. The projects are developed by working together with many of the companies and developers that are behind these exciting technologies.

- JPA
- MongoDB
- redis
- neo4j
- jdbc
- couchbase (community)
- elasticsearch (community)
- cassandra (community)
- dynamodb (community)

2.3.3 PlayORM

Playorm CITE is an open-source library developed by buffalo software built to speed up developer productivity of developing a NoSQL scalable solution.

supports Cassandra, MongoDB and HBase. similar concepts as for JPA but custom implementation -¿ custom annotations

2.3.4 Couchbase UnQL

UNQL started with quite some hype last year. However, 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. IMHO one of the major issues with the current UNQL is, that it tries to cover everything in NoSQL, from key-value stores to document-stores to graph-database.

And here I think is where UNQL is totally right. We need something similar for the NoSQL world. But it should not try to be a fits all situation.

July 29, 2011 Couchbase, the leading NoSQL database company, and SQLite, maker of the worlds most widely deployed SQL database engine, today announced the release into the public domain of a jointly developed NoSQL query language. Unstructured Data Query Language, or UnQL (pronounced Uncle), is a collaborative effort to bring a familiar and standardized data definition and manipulation language to the NoSQL domain. Both Couchbase and SQLite have committed to delivering products that embody the language.

Created by CouchDB creator Damien Katz and SQLite creator Richard Hipp, UnQL extends aspects of SQL to NoSQL databases. According to Phillips, its an expressive language that, like SQL, lets the database do heavy lifting instead of putting the burden on application developers to write certain functionalities into the application.

2.3.5 SOS Platform

2.4 Cloud Platform Independent Model

2.5 Summary

In this chapter has been introduced some of the main reasons that leads to the NoSQL database definition and why industry is so interested in those kind of technology. NoSQL technology has been compared with SQL systems to highlight the deep differences but especially the lack of a common language definition. Have been discussed some of the major project that try to define a common language among different NoSQL solution. Finally has been presented

the CPIM library, a more general approach in a common language definition in PaaS environment.

Chapter 3

Problem setting

- why use kundera among others
  - open source
  - JPA standard interface
  - community
  - many supported databases
  - polyglot persistence
  - client extension framework
  - used in production
- why include it into CPIM
- why extend CPIM to include migration
  - vendor lock-in and thus costs
  - cost of offline migration (shutdown, migration, restart)
  - live data synchronization and migration
  - functionality (for map reduce job better persist over hbase)

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 [4] 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)

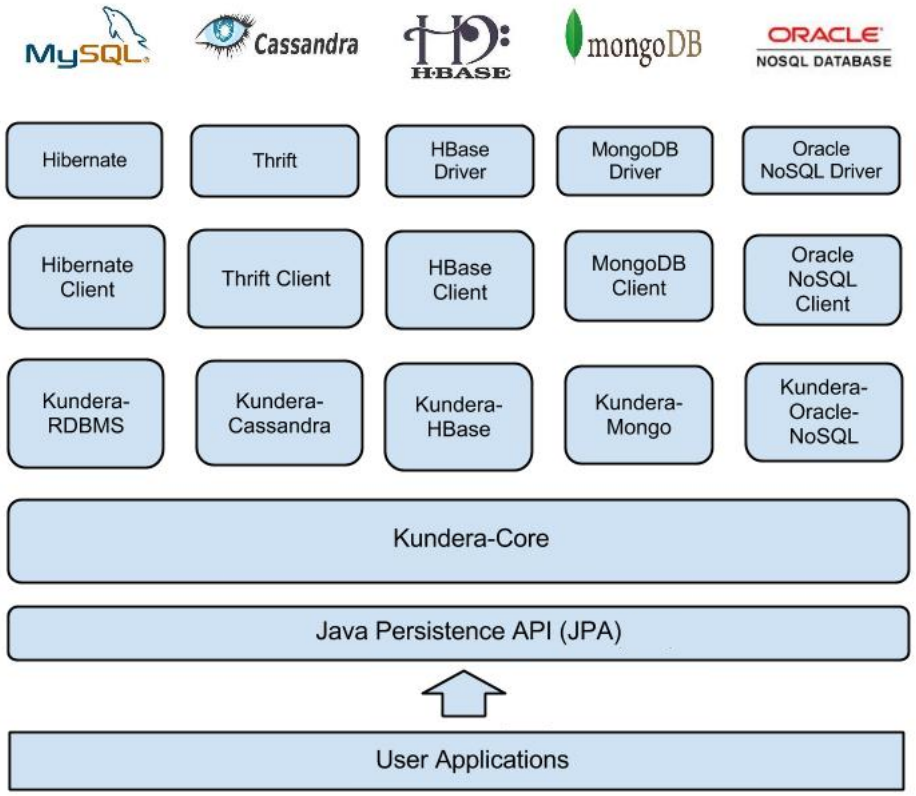

Figure 4.1: Kundera architecture

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 `Client Factory` 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 `Client Factory`, which is used by Kundera to instantiate the new Client;
- the `Query implementor`, which is used by Kundera to run JPA queries by invoking appropriate methods in Entity Readers;
- the `Entity Reader`, which is used by Kundera to translate the queries into correct client method calls;
- optionally the `Schema Manager`, 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 `Query Implementor`. Unfortunately no help was given by the community about the issue on methods arguments. Hence the most valid solution to approach the extension development 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, have to create an object that can be persisted by reading all the entity meta-data given as arguments and looking as 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, each feature developed is described in a dedicated section

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.

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 by *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 [3] 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 Datastore 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 strong consistency. 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 [7], 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 entity, 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 from Datastore the entities.

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. Is so impossible, unless asking to the user, to decide which relation use to hierarchically organize entities

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, 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 other Kundera client implementation and with the classic implementations (as well as relational databases client implementations).
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     @Column(name = "SALARY")
13     private Long salary;
14
15     /* an employee have one and only one phone */
16     @OneToOne
17     @JoinColumn(name = "PHONE_ID")
18     private Phone phone;
19 }
20
21 @Entity
22 @Table(name = "Phone", schema = "keyspace@pu")
23 public class Phone {
24     @Id
25     @GeneratedValue(strategy = GenerationType.AUTO)
26     @Column(name = "PHONE_ID")
27     private String id;
28
29     @Column(name = "NUMBER")
30     private Long number;
31 }
```

Listing 4.2: Example entities

Queries

Kundera queries are to be expressed in JPQL; the standard JPA query language, which is a object oriented query language based on SQL [5]. Kundera supports all of the clauses of JPQL, but with some restrictions since clauses

PHONE_ID	NUMBER
3cb26744	123456789

(a) PHONE

EMPLOYEE_ID	NAME	SALARY	PHONE_ID
112b18e7	Fabio	123	**3cb26744**

(b) EMPLOYEE

Table 4.1: Mapping of entity fields on Google Datastore

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 JPQL queries so almost all the clauses are supported except for the *LIKE* clause.
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 [7]) provide the user with 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 *Embedded Entity*. 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`, `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.

For a complete reference of Google Datastore extension configuration see 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 set eventual by default.
Since both partition-key and row-key support only 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 to are available:

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, 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 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 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.

PHONE_ID	**NUMBER**
DEFAULT_3cb26744	123456789

(a) PHONE

EMPLOYEE_ID	**NAME**	**SALARY**	**PHONE_ID**
DEFAULT_112b18e7	Fabio	123	**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 *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 even collections or maps composed of complex data types, the simplest solution is to serialize the entire collection (or map) to a `byte` array and, when persisting the entity. When reading back from the database, the entity isde-serialized properly.

Enum fields

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

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 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 the table is created 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 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.
So 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 *HTTP* 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 Google Datastore and 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

In this chapter will be presented the work made for CPIM extension. In section 5.2 and 5.3 is described the previous state for the NoSQL service of CPIM and how it has been modified to integrate Kundera as unique persistence provider and what problem has been faced during the process.
From section 5.4 are described the various part developed for *Hegira* integration, what are the feature supported by this integration and which design choices has been put in place.

5.2 CPIM architecture

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) 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 the first call to `MF.getFactory()` which read the configuration files and build an instance of the `CloudMetadata` class which will be referenced by all the other services and contains all the information stored in the configuration files.

The library is organized in several packages each of one 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, is instantiated one service implementation or another.

5.2.1 NoSQL service

The architecture of the NoSQL service before this work has been reported in figure 5.1.

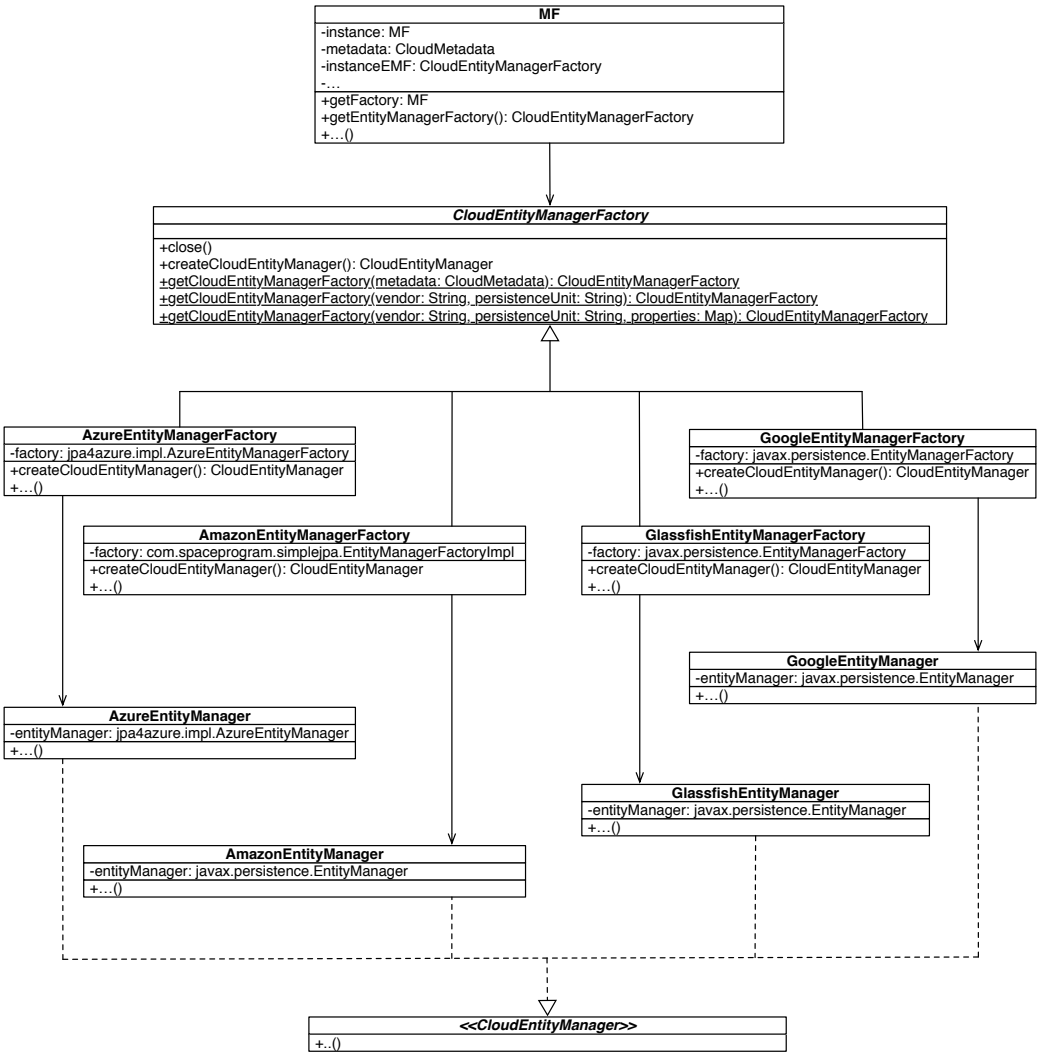

Figure 5.1: NoSQL service architecture

To use the service, the first step is instantiate a `CloudEntityManagerFactory` and, depending on the configuration file, this factory instantiate the vendor specific factory. For example in case that Google is the chosen 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 the 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.

JPA is not a default language for NoSQL but, due to its wide usage among Java developers, several JPA implementation has been build 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 provider in CPIM, one for each cloud provider:

- for *Google Datastore* its used an official JPA implementation, available inside the SDK;
- for *Amazon SimpleDB* its used **SimpleJPA**, a third-party implementation of the JPA interface;
- for *Azure Tables* its used **jpa4azure**, a third-party implementation of the JPA interface.

There are couple of things to notice: Amazon SimpleDB has been deprecated in favor of DynamoDB and *jpa4azure* is not being maintained anymore, therefore CPIM 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 rely on to provide the NoSQL service, the proposed solution 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 will define what datastore is interested in. Another benefit of this architecture is that the choice of the NoSQL technology is no more bound to the vendor specified in the CPIM configuration file, is in fact possible deploy the application in one of the supported PaaS provider and choose as NoSQL solution of another one which will be addressed remotely simply by configuring the *persistence.xml* Moreover its possible exploiting the Kundera polyglot persistency, to persist part of the data in a database and another part in another one defining the persistence units accordingly.

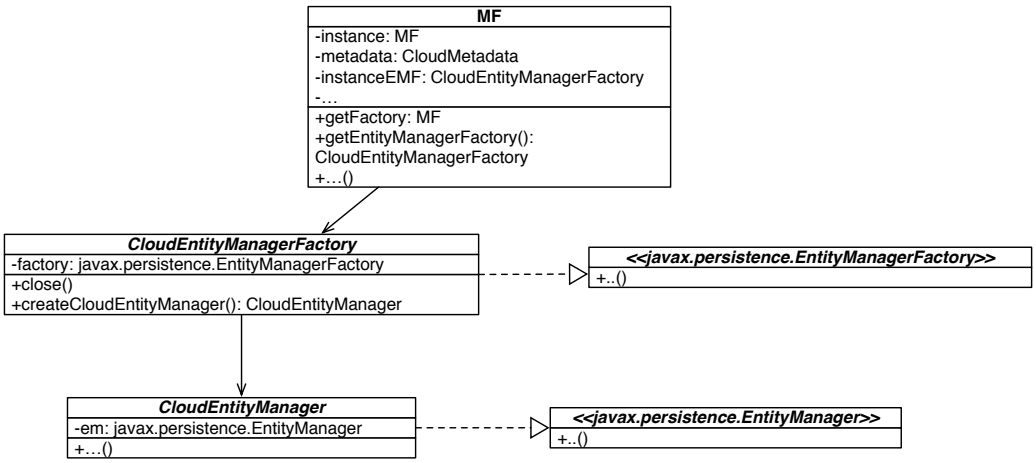

Figure 5.2: The modified NoSQL service architecture

The actual implementation is completely provider agnostic in the sense that actually Kundera is not required as dependency and in fact is not listed as a dependency for CPIM. At run-time, when a Kundera client will be listed in the dependency of the user application, as well as CPIM, the persistence provider dependency will be satisfied.
This provider agnostic implementation is due to the fact that the `CloudEntityManagerFactory` and the `CloudEntityManager` implements

respectively the JPA interfaces `EntityManagerFactory` and `EntityManager`. The actual call to the run-time provider is within the `CloudEntityManager` that on construction instantiate an instance of the provider `EntityManager` and uses that reference to delegate every method execution to it.
This can seems a over-designed architecture but it turns out to be extremely necessary in order to provides 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 provider, the desired database is defined in the *persistence.xml* through the Kundera client selection. For this reasons all the old libraries that provides a JPA implementation for a specific provider can be removed from the CPIM. This tentative of cleaning the dependency of 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 problem when multiple persistence provider are found in the classpath and has not be found a way to force the selection of Kundera as persistence provider (besides specifying it in the *persistence.xml* file).

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

- `jpa4azure`, which contains the code that implement the JPA interface;

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

The `jpa4azure` package has been removed and the library rebuild 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.

CPIM shows more errors of the code in the Queue service and after some investigations, turns out that when *jpa4azure* was extended the class `AzureQueueManagerFactory` and other were introduced. The problem was that `AzureQueueManagerFactory` use the JPA interface to communicate with the Queue service so removing the support to JPA interface we have lost the support to Azure Queue service. A solution to this would be rewrite the CPIM Queue services 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** [6]. An high level schema of the interaction we want to achieve is reported in figure 5.3

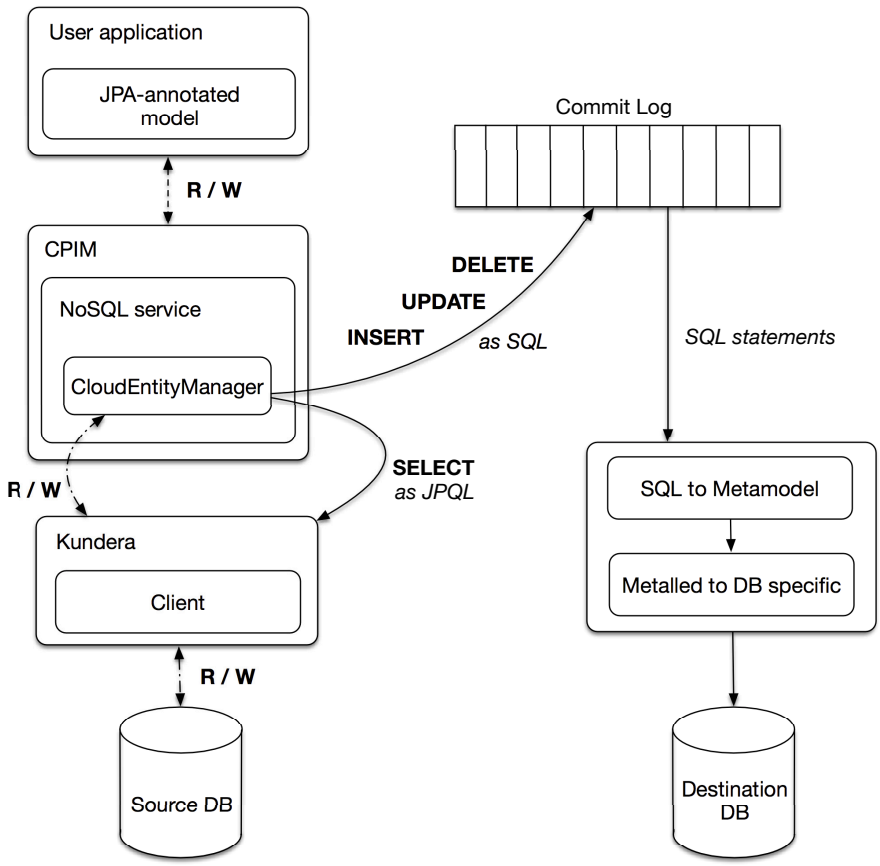

Figure 5.3: High level schema of interaction

In the above 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.
CPIM library needs to connect to the migration system in order to understand when migration is in progress and in that case only bypass the interaction with Kundera by building a string representation of the user operation as a SQL statement, then this string is sent to the commit log of *Hegira* that then will pop the statements and translate then 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.

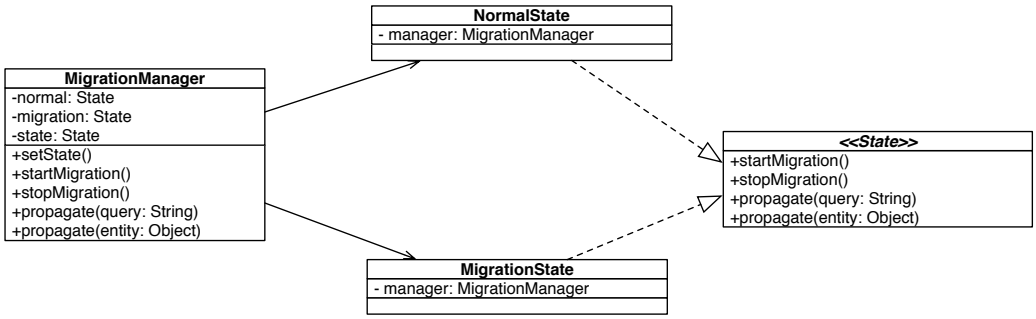

Figure 5.4: `MigrationManager` class diagram

The pattern permits to the `MigrationManager` to delegates 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.

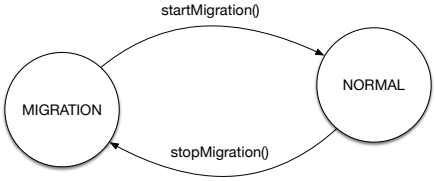

Figure 5.5: `MigrationManager` states

This part of the design was actually made before knowing how the interactions will exactly be since the component to interact with were not complete yet. Hence in order to have a well-defined place in which behavior has to be encapsulated, the state pattern was the best solution even for future extensibility in case the interaction with the migration system will become more complex.

5.5 Intercept user operations

The first operation that needs to be analyzed is where is possible to intercept user operation is a way that is completely transparent to the user. The operation that we want to intercept are the insert, update and delete operation cause those are the operation that alter the structure of the data and thus are the one that needs to be processed by the migration system.

5.5.1 Intercepting CRUD operations

CRUD operation are handled by the `EntityManager`, three are the methods that needs to be intercepted:

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

User does not invoke methods directly on the provider entity manager but interacts with the persistence provider through the `CloudEntityManager` class. Without the support for the migration system the `CloudEntityManager`, as stated previously, delegates every method call to the provider entity manager, to integrate migration and synchronization logic, the methods mentioned above should contain a little more amount of logic shown in the snippet of code 5.1 taking as 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 migration the provider is bypassed and is visible a call to the `propagate` method, it accept two arguments: the entity to be converted to a 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 `propagate` method of the migration state is responsible of building the requested statements using the statement builders and then sending the generated statements to the commit log of *Hegira*. Both action are described in detail in the following sections.

5.5.2 Intercepting queries

Looking at the JPQL specification [5] turns out that JPQL does not support *INSERT* statements and so the only way user have to persist entities is through `EntityManager.persist(Object entity)` that is one of the case described in the previous section, so only the remaining cases (*UPDATE* and *DELETE*) needs to be intercepted as query.
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 and thus from the migration system because there are not so many storage that provides a SQL-like language to specify queries; `createQuery` and `createNamedQuery` are supported, instead query creation through `CriteriaQuery` is currently not supported.

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 is performed in the query creation method in both the versions that return an instance of `Query` and `TypedQuery`. The actual JPA query generation is delegated to the persistence provider then, 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 giving 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, maintains principally a mapping between table names and their class canonical name (full package plus the class name). The content of this class is built the first time is queried (since is a singleton instance) and does not read directly 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. 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 Data synchronization

In this section is faced the problem of synchronization that allows the migration to be performed live.
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 identifier of that entity since the insert operation have already persisted the entity into the underlying database and thus generated the identifier. Since we want to guarantee a synchronization with the migration system, user cannot define its own identifiers but them needs to be assigned from the migration system. The main caveats is that such assignment has 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 interacts with the synchronization service of Hegira and handle the *sequence numbers* i.e. the entities identifiers defined by Hegira to achieve 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

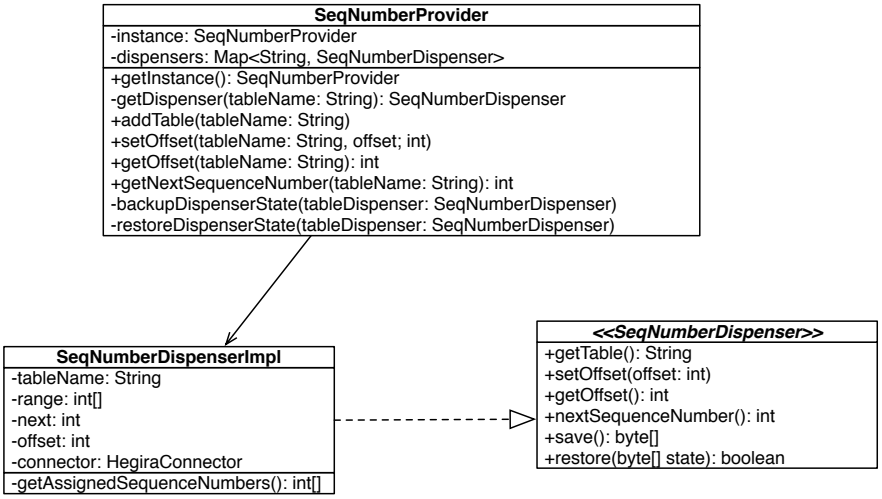

Figure 5.6: Sequence numbers handling architecture

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

1. provide a unique access point where requesting the next assigned sequence number for a table

2. initialize or restore the state of the dispenser for each of the persisted tables

The first 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 dispenser.
In case the user wants to have control over the sequence number range for a particular table, it has been made configurable at run-time. At construction time each `SeqNumberDispenserImpl` class reads the default range from `CloudMetadata` but by calling the method `setOffset(String tableName, int offset)` on the `SeqNumberProvider`, will be changed the offset of the `SeqNumberDispenser` responsible of the given table.

The second functionality is achieved by requesting to the `SeqNumberDispenser`(s) their state representation (as a `byte` array) by calling the method `save()` on the dispensers and then saving it to a Blob storage or to file depending to the configuration specified inside *migration.xml*, 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 its state representation to restore. In this mechanism the `SeqNumberProvider` is completely agnostic to the actual state representation chosen by the `SeqNumberDispenser`(s), this design choice has been made to make future extensibility more easy and less constrained.
The list of all the tables to be persisted is retrieved from the `PersistenceMetadata` mentioned previously for named queries.

5.6.1 Contacting the synchronization system

The interaction with the synchronization system as now was only described as a method call. Those calls are made on an external library (`zkWrapper`) that connects to a zookeeper instance to communicate with the synchronization system and receive the assigned sequence numbers. Since the zookeeper library issues threads to handle communication, was not possible to use this library for Google App Engine since the App Engine run-time does not permit to spawn thread. Two are the feature that requires to communicate with the synchronization system and so use the `zkWrapper` library:

- the migration state listener that modify the `MigrationManager` state accordingly;

- the `SeqNumberDispenser`(s) that needs to retrieve the sequence number assigned to tables.

The solution adopted was to modify the `zkWrapper` library to include an HTTP version that handle 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

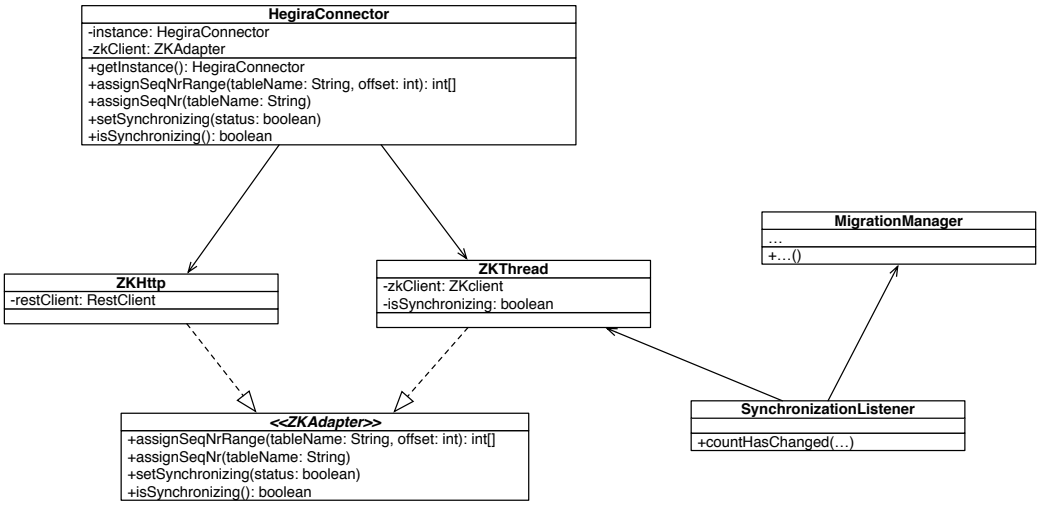

Figure 5.7: Contacting the synchronization system

The `HegiraConnector` is the class responsible of deciding which kind of client needs to be instantiated reading the configuration parsed in `CloudMetadata`. 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 deploy the application on a thread-capable client and configure the *migration.xml* accordingly, an instance of `ZKThread` is built. This version of the client uses directly the implementation of the

library `zkWrapper` since there should not be any problem in thread spawning. The `isSynchronizing()` methods 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 migration state change.

HTTP-based client In case that threads are not supported by the cloud provider 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 and asynchronously notified of a change in the migration state and is not possible to somehow cache the state or make assumption on it, each call of the `MigrationManager` to the method `isSynchronizing()` will perform an API call to the remote server and will return the state of the synchronization just queried.

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

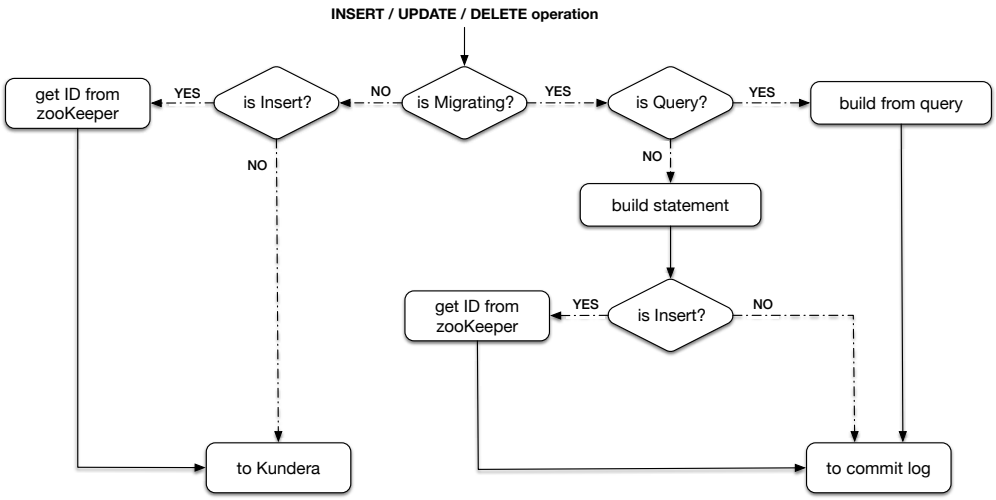

Figure 5.8: Interaction flow chart

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 statements to be sent to *Hegira*.

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

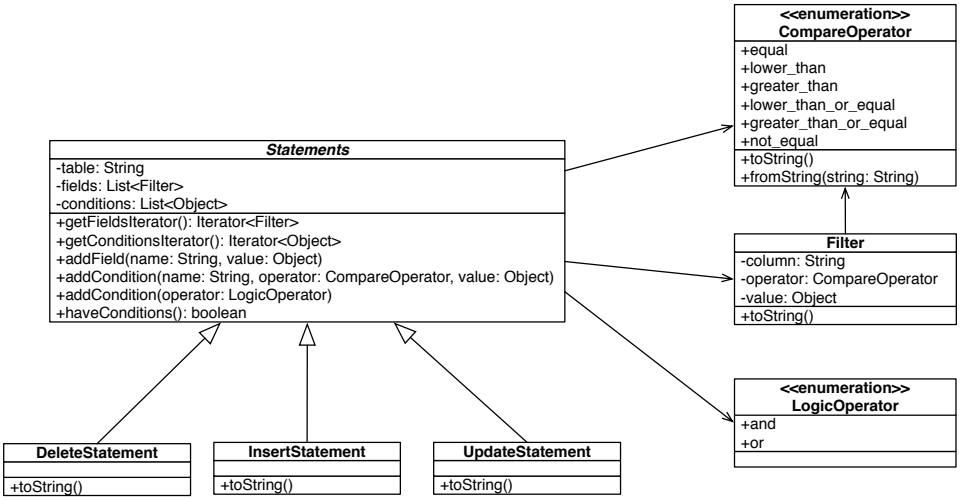

Figure 5.9: Statements structure

The `Statement` class maintains three main fields:

- `table`, that contains the table which the statements refer to;
- `fields`, a list of element of class `Filter` that contains the elements presents in the *SET* clause in case of *UPDATE* statements or the inserted values in case of *INSERT* statements;
- `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* statements 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 the exception is thrown trying to add to it a *field*.

Defined the statements structure is then necessary to provide a way to build the correct instance of 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.

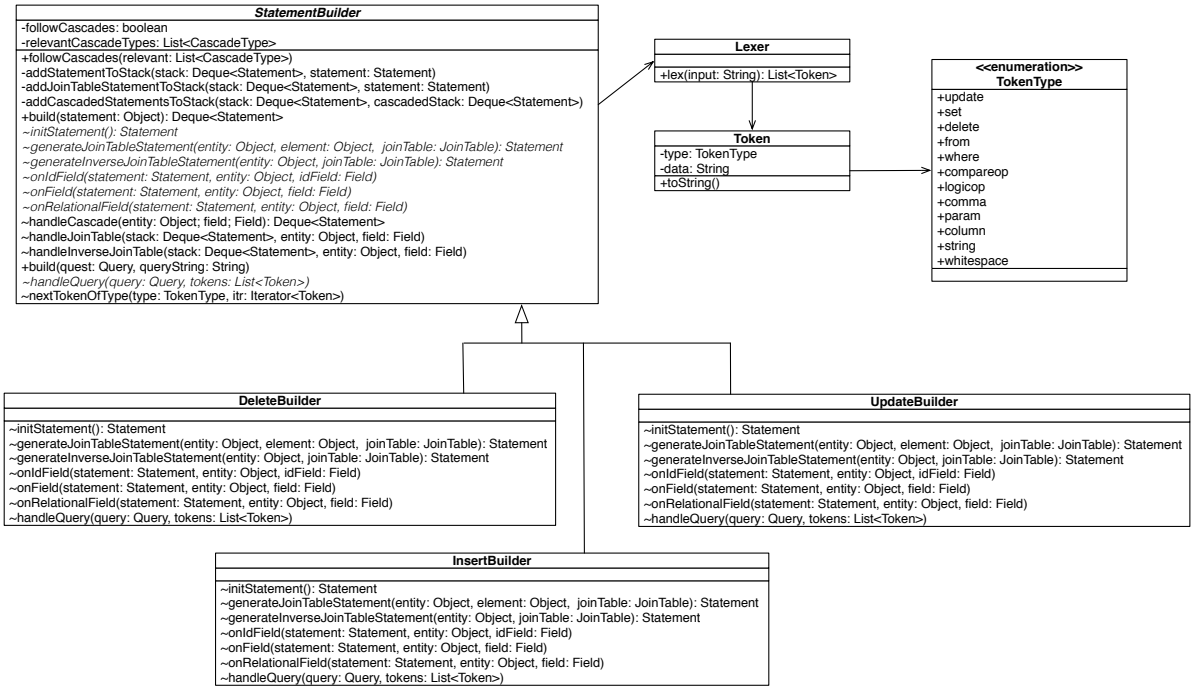

Figure 5.10: Statement builders

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 statements 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 define some *abstract* methods that are statement-specific and han-

dled 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, this allow to the `StatementBuilder` to acts independently from the specific statement type and then 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 problem in generating statements form objects were the cascade type. From the JPA specification [5], 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, for example if an entity A is inserted and is related to another new entity `B`, the entity `B` must be inserted after the insert of the first entity. Even if it's possible to guarantee this ordering, the main problem is that the migration system cannot guarantee operation orderings since such operations should be executed within a transaction.

Even if the migration system does not support transactions yet, has been decided to implement also the case of following the cascade types in case the migration system will be able to guarantee operation ordering in future versions. This behavior has been made configurable, the default value can be changed through the relative voice in the `migration.xml` (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, asks to that class in order to decide if the build process should or not consider the cascade types and if its the case they set the `relevantCascadeTypes` accordingly. The relevant cascade types has 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

To define the statements ordering has been taken the ordering that should be respected as if the statements are for a SQL database. In this situation *join table statements* must be taken with particular care since inserts in the join table must be happen **after** the insert of the entity itself and deletes in the join table must be happen **before** the delete of the entity itself.

The builder abstract class `StatementBuilder` provides a single entry point for statements building which is the method `build(Object entity)`. This method is designed following a template pattern, the designed general algorithm (reported in algorithm 1) perform all the operations needed to build the statement and 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.
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 recursively calls `build(Object entity)` passing as `entity` the related object.

5.7.2 Build statements from JPQL queries

Cascade modification for *UPDATE* and *DELETE* statements should raise if the statement, beside modifying the entity directly stated in the query, modify another related entity. Fortunately this is not a possible case for JPQL, for the

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
```

object case this is possible since objects keeps reference to the related entity and if that has been modified, a cascade operation should be performed.

The main problem for JPQL queries is parsing. Since JPQL is a object query language it make use of an object identifier on which use 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 following snippet:

```
# 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 5.4: JPQL to SQL translation

The parameter mapping can be done easily since `CloudQuery` and `TypedCloudQuery` by implementing the required JPA interfaces can intercept the `setParameter` method and so maintain the name-value mapping.

To solve the parsing problem, the Kundera code was inspected to understand how JPQL queries are parsed but turns 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 tokenize 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 tokenize the query using the lexer and then call the abstract method `handleQuery(query, tokens)` that is implemented in `DeleteBuilder` and in `UpdateBuilder` which are responsible of iterate 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.
The statement stack is iterated and each time a statement is removed from the head of the stack (LIFO order). Each one of the extracted statement is then sent to `Hegira`.

5.8 Interoperability of stored data

The Kundera client developed and described in chapter 4 was developed to be as much as consistent to the other client developed for Kundera to be more

likely accepted by the community and so are not completely 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 to the application without changes besides the new database configuration.
The problem for Kundera clients are the relationships. Each database have its own ways to define identifier for the persisted entities, for Google Datastore there's the `Key` with *Kind* and an *identifier*, for Azure there are the *partition-key* and the *row-key*. 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 require a good amount of changes in the migration system, other solution 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's not the case either in Datastore or Azure Table. The pair <*column family*, *key*>is sufficient to identify an entity (composed by one or more columns), so is needed a mapping between database-specific terminology to the more general one, this mapping is shown in table 5.1.

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.

At this point is needed a common way of persisting relationships as column family and key in a way that is interoperable among both the client extension. The proposed solution is to persist `columnFamily_key`. This solution has to

be preferred w.r.t the one that require modification of the migration system since the interoperability is achieved transparently to 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 is not providing the class of the entities involved in the join table but just 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 find operation. To be more consistent and 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 Table The Azure Table extension has been modified too to reflect the newly defined standard for relationships. In Azure Table 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 is not a possibility that can be guaranteed since if an entity is persisted with a partition-key, it will be read by the Datastore extension as of that *Kind*. Since the `Kind` in Datastore extension is the entity table name, has been decided to lock the Azure Table partition-key to the table name so user cannot decide its own since this will break the interoperability of data.
The same discussion for the join tables made previously for Datastore applies as is also 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 has been described how was possible to integrate Kundera as unique persistence provider in the NoSQL service and the problem encountered in the process.
From section 5.4 has been described the general interaction we wanted to build to make CPIM and Hegira communicate and was then introduced and described the architecture and the design choices operated in order to develop such interaction.

Chapter 6

Evaluation

6.1 Introduction

Introduzione agli argomenti trattati nel capitolo, dalle 4 alle 10 righe.

6.2 Test correctness of CRUD operations

JUnit tests

6.3 Performance tests

Task about YCSB and Kundera-benchmarks

6.4 Data generation

app that generate data for migration test

6.5 Summary

Riassunto del capitolo

Chapter 7

Conclusions and future Works

Conclusioni del lavoro e sviluppi futuri. Massimo una o due pagine.

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

[4] Kundera. https://github.com/impetus-opensource/Kundera, 2015. [Online].

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

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

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