

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	xv
List of Tables	1
1 Introduction	3
2 State of the art	5
2.1 Introduction	5
2.2 NoSQL databases	6
2.2.1 NoSQL motivations	6
2.2.2 NoSQL characteristics	6
2.2.3 Standard language	6
2.3 Approaches for a common language	6
2.3.1 Kundera	6
2.3.2 Spring-data	6
2.3.3 Espresso Logic	6
2.3.4 Couchbase UnQL	6
2.3.5 PlayORM	6
2.3.6 ? Apache Gora	6
2.3.7 ? Apache Phoenix	6
2.3.8 ? SOS Platform	6
2.4 Cloud Platform Independent Model	6
2.5 Summary	6
3 Problem setting	7
4 Kundera extension	9
4.1 Introduction	9

TABLE OF CONTENTS

4.2	Overview of Kundera	9
4.2.1	Kundera's Client Extension Framework	11
4.2.2	Approaching the extension	11
4.3	Developing client extensions	12
4.3.1	Google App Engine Datastore client	13
4.3.2	Azure Table client	19
4.4	Summary	22
5	CPIM extension	25
5.1	Introduction	25
5.2	CPIM architecture	25
5.2.1	NoSQL service	26
5.3	Kundera integration	28
5.3.1	Problems encountered	29
5.4	Hegira integration	30
5.4.1	Migration Manager	31
5.5	Intercept user operations	32
5.5.1	Intercepting CRUD operations	32
5.5.2	Intercepting queries	33
5.6	Data synchronization	35
5.6.1	Contacting the synchronization system	37
5.7	Build statements from user operations	39
5.7.1	Build statements from objects	42
5.7.2	Build statements from JPQL queries	44
5.7.3	Sending statements to Hegira	45
5.8	Interoperability of stored data	45
5.8.1	Kundera clients modification	46
5.9	Summary	47
6	Evaluation	49
6.1	Introduction	49
6.2	Test correctness of CRUD operations	49
6.3	Performance tests	49
6.4	Summary	49
7	Conclusions and future Works	51

Appendices	53
A Configuring Kundera extensions	55
A.1 Introduction	55
A.2 Common configuration	55
A.3 GAE Datastore	56
A.4 Azure Table	58
B Configuring CPIM migration	61
B.1 Introduction	61
B.2 <i>migration.xml</i>	61
B.2.1 Configure the ZooKeeper client	62
B.2.2 Configure a sequence number backup	63
B.3 Use CPIM without migration system	64
C Run YCSB tests	65
C.1 Introduction	65
C.2 Preliminary operations	65
C.3 Run tests for low-level API version	66
C.3.1 Property files	66
C.4 Run tests for Kundera version	68
C.4.1 <i>persistence.xml</i> configuration	68
Bibliography	69

List of Figures

4.1	Kundera architecture	10
5.1	NoSQL service architecture	27
5.2	The modified NoSQL service architecture	28
5.3	High level schema of interaction	30
5.4	MigrationManager class diagram	31
5.5	MigrationManager states	31
5.6	Sequence numbers handling architecture	36
5.7	Contacting the synchronization system	38
5.8	Interaction flow chart	40
5.9	Statements structure	40
5.10	Statement builders	41

List of Tables

4.1	JPQL clauses support for the developed extension	23
5.1	Column family and Key mapping among supported databases. .	46

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 ...
- In Chapter ...
- In Chapter ...
- ...

Finally, in Chapter 7, ...

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

2.3 Approaches for a common language

2.3.1 Kundera

2.3.2 Spring-data

2.3.3 Espresso Logic

2.3.4 Couchbase UnQL

2.3.5 PlayORM

2.3.6 ? Apache Gora

2.3.7 ? Apache Phoenix

2.3.8 ? 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

In this chapter will be presented briefly the 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.

Then are discussed the detail of the two developed Kundera extension, in section 4.3.1 the one for Google Datastore and in section 4.3.2 the one for Azure Table.

4.2 Overview of Kundera

Kundera [4] is an implementation of the JPA interface that now 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.

The 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)

Kundera extension

- Redis (version 2.8.5)
- Neo4j (version 1.8.1)
- CouchDB (version 1.0.4)
- Elastic Search (version 1.4.2)

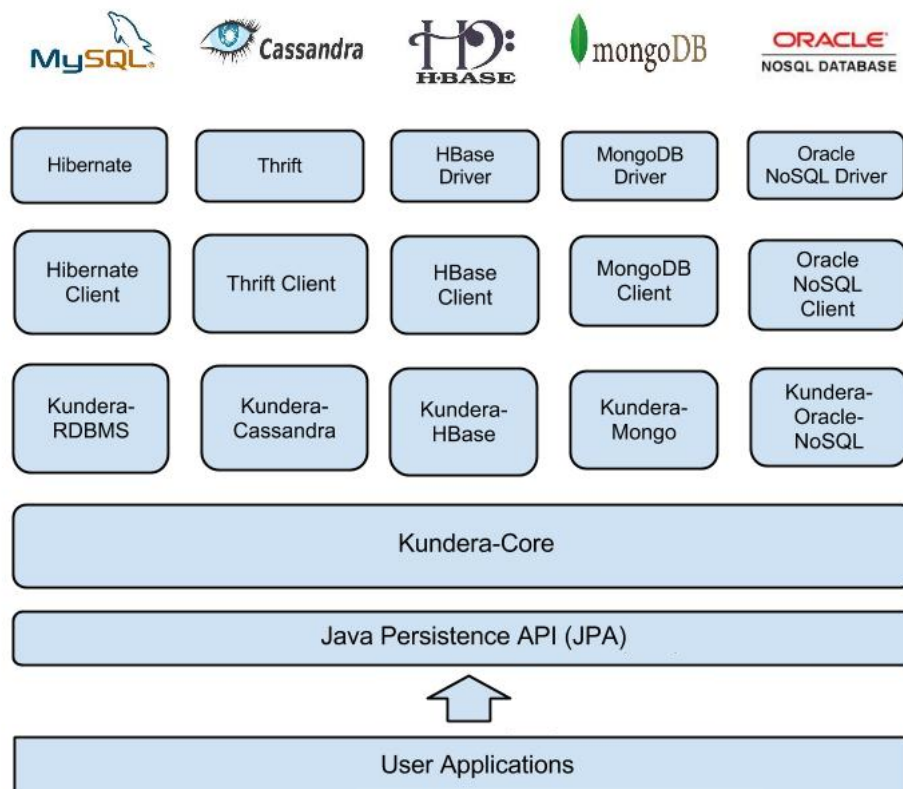


Figure 4.1: Kundera architecture

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 try to offer a common way to interacts with NoSQL databases through a well defined interface and as on open source project make other developers able in using and extending it, adding support to other databases. Is so available a *Client Extension Framework* described in the Kundera documentation which provides a short description about how Kundera clients should work and provides 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 queries
- the `Client Factory`, which is used by Kundera to instantiate the `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

It all seems well structured but the problem is that the documentation is actually outdated. 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 skip completely to describe what kind of information are carried by the argument of the methods that needs to be implemented

Due to those problems, the solution was to write on the Kundera Google group page to ask the community for more updated information about Kundera

extension. Briefly an answer has come and I've started a conversation with one of the developers of Kundera who helped me giving the updated information for the Kundera's *Client Extension Framework* and tell me to look forward to the other client implementation for some examples. Thanks to the updated information 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**.

At this point since no answer were given about the information carried by the methods arguments, the most valid solution was to approach the extension in a test driven way trying so to reverse engineer the arguments.

Looking at the tests code of the other clients, I've written a set of unit tests one for each feature I was planning to support (tests are analyzed in chapter 6). By doing this and by looking at the other client implementation I was finally able to understand the structure of the method arguments, they carry information about the operation to be performed and about the entity on which the entity needs to be performed. Entity information are structured in a data structure filled by Kundera by parsing the annotation that the user define on the entity class, those meta-data includes for example the table name, the column names and the relationships in which the entity is involved.

With the client structure defined, the tests written, the knowledge of the responsibilities of each method and of the Kundera meta-model, was then possible to begin the actual extension developing.

4.3 Developing client extensions

Two are the extension that have been developed for Kundera, the first one for Google Datastore and the second one for Azure Table. After the difficulties in figuring out how the extension have to be carried out, a main structure has been defined and kept for both the extension so the two projects have many parts in common. In the following sections the extensions are presented separately, each developed feature is described in a dedicated section, the general concepts are introduced as they are encountered and will be referenced further on if necessary specifying the differences if any.

4.3.1 Google App Engine Datastore client

The first extension that has been faced is the one for Google App Engine Datastore [2] the NoSQL solution available in the App Engine runtime, a key-value storage build on top of Google BigTable.

JPA identifier

In Google Datastore the most basic unit that can be stored is an *Entity* which is identified by a *Key* and composed of *Properties*. Keys contains 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 pre-defined defined set of Java data types.

The adopted solution is to handle the key internally. Each time an operation is required on Datastore the key relative to the entity is built, the key *Kind* is directly mapped to the table name and the Key identifier is the user defined id specified in the `@Id` annotation.

IDs can be specified by the user or automatically generated, there are three possibilities:

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

For each case the ID can be specified by the user before the persist operation. Since Kundera support the JPA feature for auto-generated IDs through `@GeneratedValue`, this possibility has been exploited for Datastore and so

Kundera extension

the user can annotate a **String** ID field to be auto-generated, 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 containing the keys of the entities which are parents of the given one and thus in the same entity group.

Consistency is managed through entity groups and so by defining the ancestor paths. Entities within the same entity group are managed in strong consistency, eventual consistency is used otherwise.

Datastore provide the possibility to create ancestor path by defining entities parent to other entities and is basically a task left to the user, no automated sorting or guessing is provided. Other wrapper around Datastore low-level API also leave this task to the user, for example in Objectify [7] the developer make use of an **@Parent** annotation to make the user able to specify the parent relationships and so 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, the only alternative way is trying to automatic guess the ancestor path.

An approach can be look at JPA relationships since they're clearly a good place to found information for guessing if two entity kind can be hierarchically related, so for each type of relation must be defined what solution can be adopted:

- for **One to Many** and **One to One** relationships, since there's an owning side of the relationship, the owning entity can be used as parent for every related entity.
- **Many to One** can be skipped because they are the inverse of **One to Many** so such related entities should be already organized.

- for **Many to Many**, since the relationship is handled through a join table, it does not make sense to relate the entities involved.

Also if possible this guessing was not done in the extension for two main reasons:

1. entities are not require to have a single relationship
2. entities with a parent require the parent Key to be universally identified

So for the first reason is impossible, unless asking to the user, to decide which relation use to hierarchically organize entities, furthermore for the second reason when declaring a entity parent to another is always necessary to know the Key of the parent (and thus its Kind and identified) beside the Key of the entity itself to be able to retrieve it from Datastore and for how Kundera is structured this information is not available during find operation in which Kundera provides only the table name, the identifier and the entity class.

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

JPA relationships

All the JPA supported relationships has been implemented in the client and have been implemented like they would be in a relational database. So for **One to One** and **One to Many** relationships on the owning side of the relationships a *reference* to the non-owning side entity is saved.

For **Many to One** relationships there would be two solutions:

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

The second solution has been adopted since more consistent with other Kundera client implementation and with the classic implementation of this relation type in relational systems.

Kundera extension

For **Many to Many** relationships a join table is created based on user directives specified in 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.

Queries

Queries in Kundera are supported in JPQL the 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, clauses can be applied on:

- primary key attributes (`@Id`) and column attributes (`@Column`).
- combination for primary key attribute and columns.

The JPQL query is parsed and validated by Kundera and to the `Query Implementor` are provided some meta-data extracted from the query which then needs to be read in order to build a database specific compatible query.

Datastore have on its own a very good support to queries so all the clauses are supported except for the *LIKE* clause.

To be able to execute queries on properties, Datastore needs to construct indexes upon those properties. Those indexes consumes the App Engine application quotas both to be stored and maintained. The API provides the possibility to decide upon which property an index should be maintained by using a different method when the property is added to the entity; `setProperty(String name, Object value)` is used to set a property which will be automatically indexed, `setUnindexedProperty(String name, Object value)` will be used otherwise.

Since a discriminator is needed to choose between the two methods, other wrapper around low-level API such Objectify [7] provides to the user an `@Index` annotation to be place upon the field that needs to be indexed but, as previously explained, is not convenient to add other annotation to the JPA standard, this will break interoperability. For those reasons, and to be able to actually execute the queries, all properties are set as indexed.

In table 4.1 can be found a complete list of the supported JPQL clauses for both extensions.

Embedded entities

Embedded fields are supported by the JPA [5] annotating the field that needs to be embedded with the `@Embedded` annotation and annotating the corresponding class with the `@Embeddable` annotation.

Implementation of those kind of entities is straightforward for Datastore because it supports them natively as an *Embedded Entity*. The implementation so make use of this feature translating the embeddable entity in a Datastore embeddable entity and then persist it within the parent entity.

Collection fields

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

Lists are natively supported by Datastore but are supported only if composed of primitives Java data types. To be able to save whatever kind of collection or map independently by the data type that compose it, the collection or map itself is serialized to a `byte` array when persisted and de-serialized when read. To simplify the developing, also Lists of primitive types, even if supported natively, are serialized.

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

Schema manager as required by Kundera has to exploit four operations:

- *validate*, which validates the persisted schema based on entity definition

Kundera extension

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

The first two cases are quite useless for a Datastore since there's no fixed schema for entities, entities with same *Kind* can have different properties without restriction. Also the *create* case is meaningless for Datastore since if a new entity of an unknown *Kind* is persisted it's created without the need of explicitly define 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 need to follow a simple structure. This file is referenced inside the `persistence.xml` and 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 have to issue implicit transactions

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

For a complete reference for Google Datastore extension configuration see the appendix A.3.

4.3.2 Azure Table client

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

JPA identifier

In Azure Table an entity to be persisted must either implement a special interface `TableServiceEntity` or be translated into a `DynamicEntity` which is basically a property container. 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 while entities are stored within the same partition key otherwise consistency will be eventual.

Since both partition-key and row-key are supported only in field of type `String` and since the JPA annotation `@Id` can be declared only on one field per class, partition-key and row-key are concatenated in a single `String` and handled internally by the extension through the class `AzureTableKey` built *ad hoc* since for Azure Table there's no a class similar to `Key` of Datastore that encapsulate both the partition-key and the row-key. This way the user have complete control over partition-key and row-key and thus on the consistency mechanism.

For the user three different approaches to handle those identifiers are available:

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

For the first case, to help the user in defining both the partition key and the row key independently by the way the extension handle them, 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`.

The second case is exploited setting the ID to a string value, this value is interpreted by the extension as the row key while the partition key is set to a default value that can be modified in the datastore specific property file

Kundera extension

described later on. Also for this case, 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 Table relationships are implemented similarly to relational systems as described previously for Datastore (4.3.1).

The only difference is that when is needed to keep a *reference* to another entity, is persisted within the entity the partition key and the row key of the related entity. Even if the pair (partition-key, row-key) is not sufficient to identify an entity universally, it is sufficient in Kundera since the information of the table is always available to the client just by looking at the meta-data of the relationship.

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.

In table 4.1 can be found a complete list of the supported JPQL clauses for both extensions.

Embedded entities

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

Collection fields

As described for Datastore (4.3.1) JPA supports collections but are not supported in Azure Tables even if composed of supported data types.

To support even complex collection or maps the simplest solution is to serialize the entire collection or map to a `byte` array when persisting the entity and de-serialize it when reading the entity from the storage.

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

Schema manager (as described in 4.3.1) have been also implemented for Azure Table and like Google Datastore, the first two cases are quite useless since there's no fixed schema and entities within the same Table can have different properties without restriction.

Azure Table need that the Table in which entities are stored exists before trying to create entities so the *create* case simply iterate over all table names and creates it 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 exists an unpredictable amount of time in which the table cannot be re-created since it still exists even if not listed. To overcome to this problem two solution can be adopted:

- catch the `StorageException` thrown when the table is created while still exists, put the process to sleep for an amount of time and then try again until success
- 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 partition key and row key for each entity in the table, this will require an high number of API call and thus an high cost of usage.

Kundera extension

So for the *create_drop* case is performed a drop of all the Tables and then 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 because the Table storage is emulated over a SQL server instance.

Datastore specific properties

As described for Datastore (4.3.1), Kundera provides a datastore specific properties file that let the user set some specific configuration.

This possibility is supported also for Azure Tables with the following available properties:

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

For a complete reference for Azure Table extension configuration see the appendix A.4.

4.4 Summary

In this chapter has been introduced in details how Kundera extension should be developed, the problem encountered during the development, how they've been addressed and the detail of the implementation of the two extensions including the feature that are currently supported.

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

Table 4.1: JPQL clauses support for the developed extension

Chapter 5

CPIM extension

5.1 Introduction

In this chapter will be presented the architecture of CPIM especially for the NoSQL service as it was before this work, then will be explained how was possible to include Kundera as persistence layer for the NoSQL service in CPIM and what problem has been faced during the process. Furthermore in section 5.4 is shown how the integration with the migration system *Hegira* has been introduced, 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 **CloudMetadata** 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

Before this work CPIM library supports NoSQL interaction through Java Persistence API. The interoperability with the supported clouds is made possible thanks to the previously described factory pattern, a complete class diagram for this service can be seen in the figure 5.1.

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 (as described in chapter 2) but, due to its wide usage among Java developers, several JPA implementation has been build upon 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. There are so three different persistence providers, one for each cloud provider:

- for *Google Datastore* its used an official JPA implementation, available inside the SDK



Figure 5.1: NoSQL service architecture

- 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 this 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 proposed solution is resumed in the architecture of 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 provider 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 it's 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 properly.

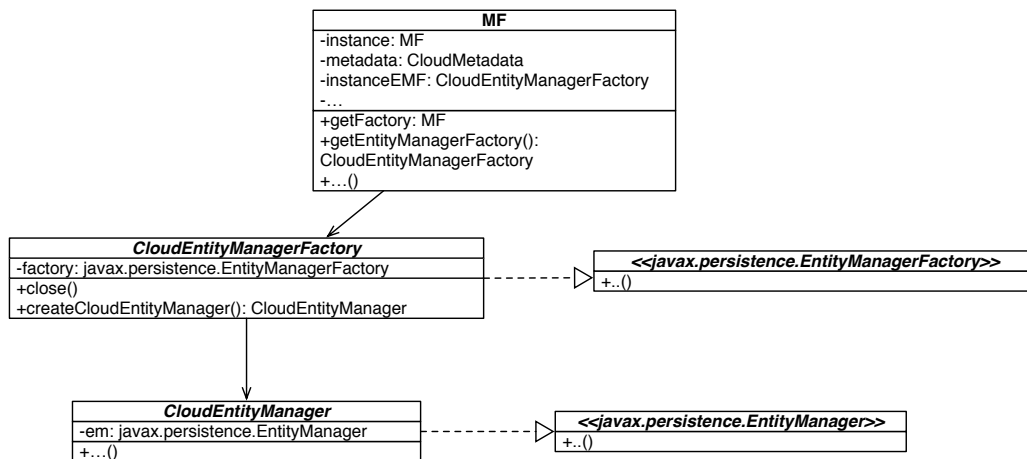


Figure 5.2: The modified NoSQL service architecture

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 per-

sistence provider dependency will be satisfied. This is due to the fact that the `CloudEntityManagerFactory` and the `CloudEntityManager` implements respectively the JPA interfaces `EntityManagerFactory` and `EntityManager`. The actual call to the runtime 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, which is somehow 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 in 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 since it was extended to support some missing functionalities of the JPA interface, the library contains two main packages:

- `jpa4azure`, which contains the code that implement the JPA
- `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 Azure SDK.

CPIM shows more errors in 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 has led to lose the support to Azure Queue service. One possible solution to this would be rewrite the CPIM Queue services for Azure using Azure SDK.

5.4 Hegira integration

To support data synchronization and migration, the NoSQL service was further modified to integrate with **Hegira** [6]. An high level schema of the interaction we want to achieve is reported in figure 5.3

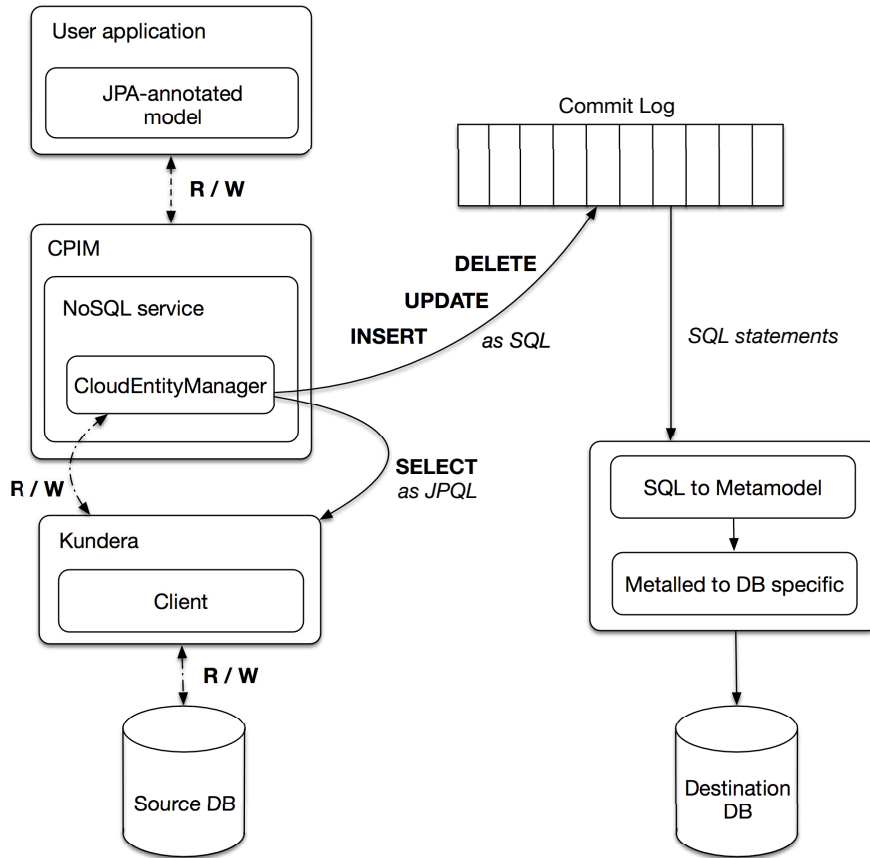


Figure 5.3: High level schema of interaction

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 and so the local database is bypassed. 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 in a SQL format, then this string is sent to the commit log of *Hegira* that then will pop the statements and translate those SQL statement 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.

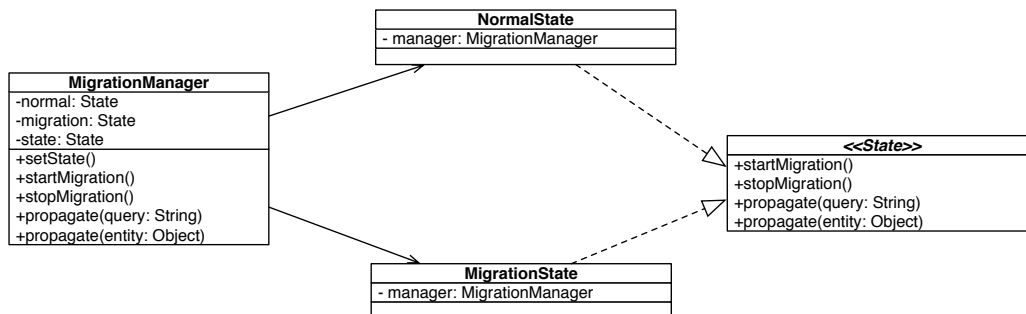


Figure 5.4: MigrationManager class diagram

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 is and composed by two states `MIGRATION` and `NORMAL` that encapsulate the required behavior.

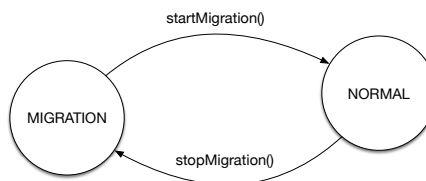


Figure 5.5: MigrationManager states

This part of the design was actually made before knowing exactly how the interactions will exactly be since the component to interact with were not finished 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 always handled in the `EntityManager`, three are the methods that needs to be intercepted:

- `EntityManager.persist(Object entity)` for the insert operation
- `EntityManager.merge(Object entity)` for the update operation
- `EntityManager.remove(Object entity)` for the delete operation

User is not directly invoking method 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 is visible the call to the **propagate** method. It accept two arguments:

- the entity to be converted to a statement
- the operation that needs to be generated

The method is called on the **EntityManager** which then delegates the execution to the current state which should be the migration state at that point. 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 later on.

5.5.2 Intercepting queries

A first look at the JPQL specification [5] revealed 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.

JPA interface provides several ways to build and execute queries all available by calling the proper methods defined in the **EntityManager** interface:

- **createQuery** from JPQL query string
- **createQuery** from an instance of **CriteriaQuery**
- **createNamedQuery**
- **createNativeQuery**

Native queries are clearly not supported by Kundera and thus from the migration system clearly because there not so many storage that provides a SQL-like

language to specify queries. Create queries through `CriteriaQuery` is currently not supported. The remaining two kind of methods are the supported ones.

JPA does not provide through the `Query` interface 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. That 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`. This allows to transparently maintains the JPQL string representation of the query. The wrapping of the persistence provider queries is performed 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 the generated query and the string representation of it (used by the provider to build the query).

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 metadata 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 reads 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 later on.

```
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 the previous section, an overview of the required logic that has been built to integrate *Hegira* was described. Now let's face the problem of synchronization that allows the migration to be performed live.

A special look needs to be reserved for 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

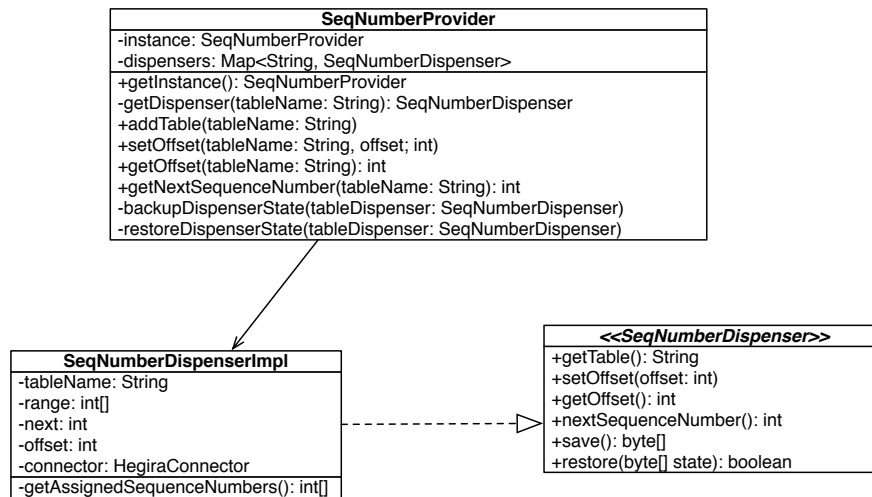


Figure 5.6: Sequence numbers handling architecture

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, maybe in the future, 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`, it will set the specified offset to 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 their state representation to restore. This mechanism in which the `SeqNumberProvider` is completely agnostic to the actual state representation of the `SeqNumberDispenser(s)` to make future extensibility more easy and less constrained. The `SeqNumberProvider` requests to `SeqNumberDispenser(s)` their sta

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 method 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`:

- 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 API 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

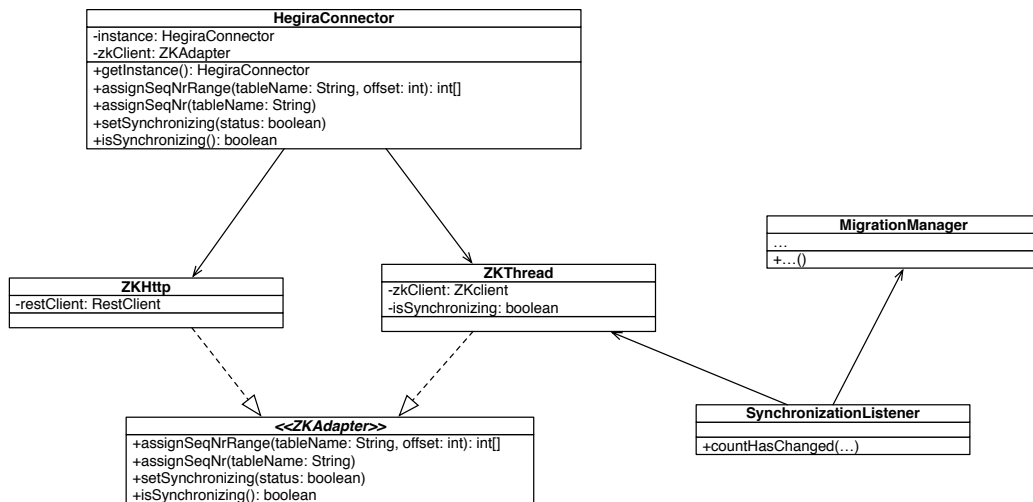


Figure 5.7: Contacting the synchronization system

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 instantiated client

5.7 Build statements from user operations

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.

5.7 Build statements from user operations

Before focusing on the statements building, since migration system interaction are getting many, it may be useful to visualize the interaction in the high-level flow chart presented in figure 5.8.

In the previous section we've focused on the sequence number retrieval, in this section we'll 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

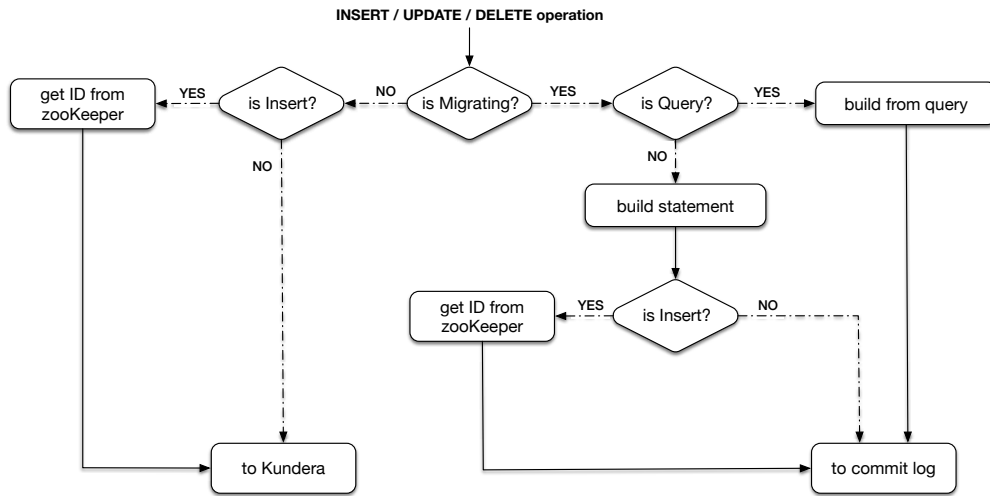


Figure 5.8: Interaction flow chart

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

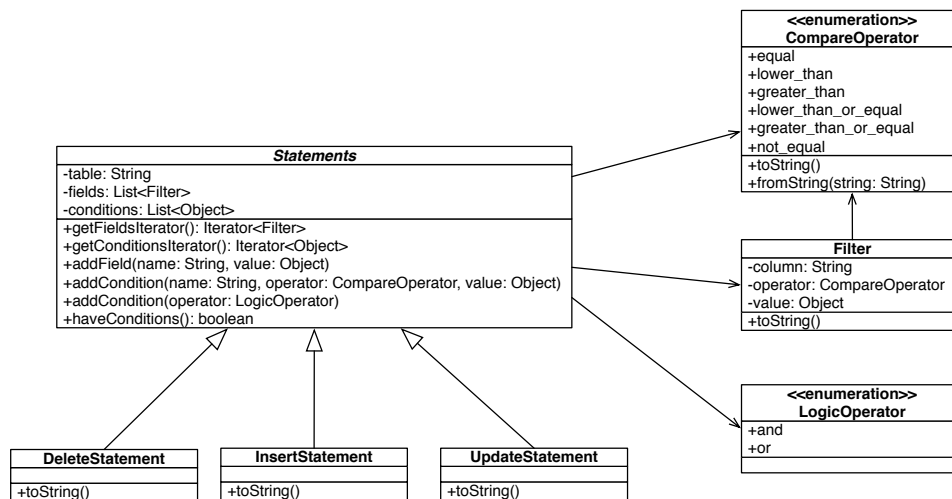


Figure 5.9: Statements structure

The `Statement` class maintains three main fields:

- `table`, that contains the table which the statements refer to

5.7 Build statements from user operations

- **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 to the statement 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 the operation on an object. To do this in an agile way a builder class has been implemented.

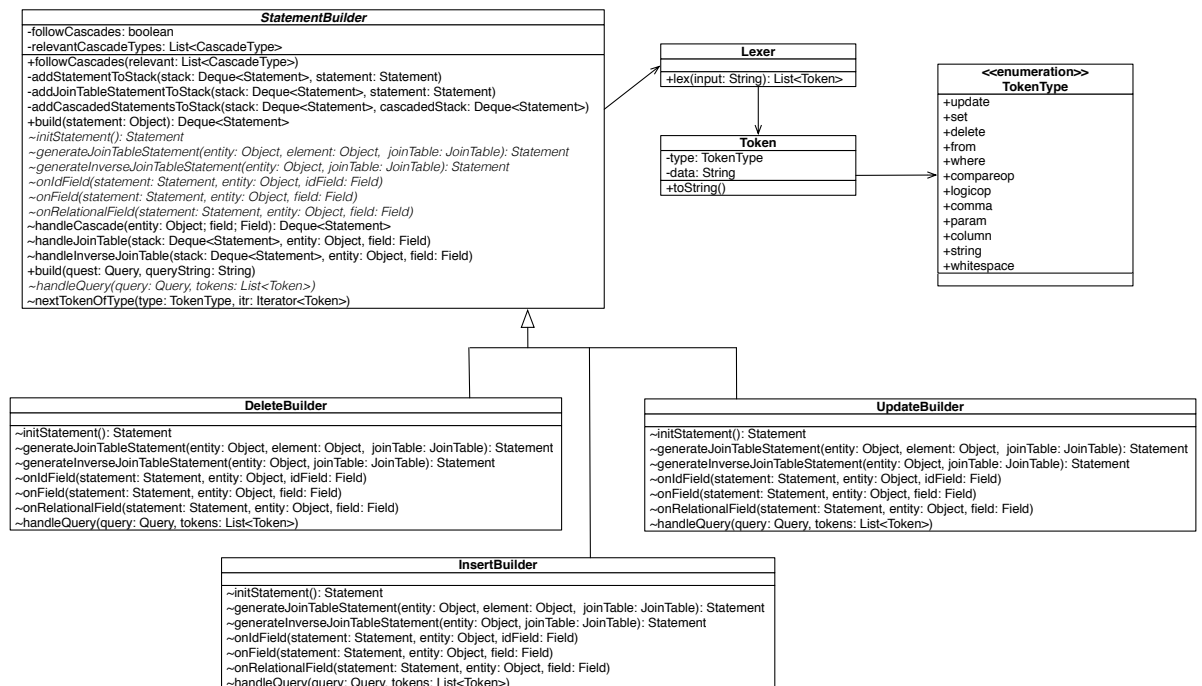


Figure 5.10: Statement builders

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 handled 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 performed 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

5.7 Build statements from user operations

problem is that the migration system cannot guarantee operation orderings since 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, 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 overridden 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 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 tokens to build the correct `Statement` instance. The `build(Query query, String queryString)` calls other various hook methods that can be overridden 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 built to be 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 database. 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 handle them. This can be done 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 possible either in Datastore or Azure Table. The pair (*column family*, *key*) is sufficient to identify an entity (composed by one or more columns) is so 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 save 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` an idea that came from the Azure Table extension which already save relationships similar to this. 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 in the find operation. To be more consistent and apply the identifier pattern (`kind.key-identifier`) even for join tables, a map is maintained in the client and built inspecting Kundera metadata 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. 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 thing to notice is that the decided pattern `partition-key-row-key` has not been broken, the only limitation is the one that fix the partition-key to the table name.

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.

Algorithm 1 Template algorithm for statements building

```
1: function BUILD(object)
2:   stack  $\leftarrow$  empty queue
3:   cascadedStack  $\leftarrow$  empty queue
4:   statement  $\leftarrow$  INITSTATEMENT
5:   SETTABLENAME(statement, object)
6:   for all field  $\leftarrow$  GETFIELDS(statement) do
7:     if ISRELATIONAL(field) & OWNRELATION(field) then
8:       if handle_cascades then
9:         cascadedStack  $\leftarrow$  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
```

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 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 that are 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 data-store specific properties that can be defined in the external datastore specific property file.

A.2 Common configuration

All the configuration is performed in the *persistence.xml* file and so 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

Configuring Kundera extensions

A name for the persistence unit is needed as it will be referenced inside the classes of the model. In the `<calss>` tag must be listed, one per tag, the full name of the classes that needs to be handled through this persistence unit. The only differences for the extensions are the kundera properties that needs to be specified inside the `<properties>` tag.

A.3 GAE Datastore

The GAE Datastore extension is available on github at <https://github.com/Arci/kundera-gae-datastore>.

The configuration is done in the *persistence.xml* file, two configuration are possible:

1. use the datastore instance within the appengine 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 appengine 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 appengine runtime 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.2: GAE Datastore emulator configuration

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 class-path, you need to specify its name 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>datastore</name>
6             <connection>
7                 <properties>
8                     <property name="..." value="..."></property>
9                 </properties>
10            </connection>
11        </dataStore>
12    </datastores>
13 </clientProperties>
```

Listing A.3: GAE Datastore - datastore specific configuration

The available properties are:

- `datastore.policy.read` [eventual|strong] default is **strong**, set the read policy.
- `datastore.deadline` optional, RPCs deadline in seconds.

Configuring Kundera extensions

- `datastore.policy.transaction` [auto|none] default is **none**, define if use implicit transaction.

A.4 Azure Table

The Azure Table extension is available on github at <https://github.com/Arci/kundera-azure-table>.

The configuration is done in the `persistence.xml` file, the properties to be specified inside the `<properties>` tag are:

- `kundera.username` *required*, the storage account name (from azure portal)
- `kundera.password` *required*, the storage account key (from azure portal)
- `kundera.client.lookup.class` *required*, and 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 class-path, you need to specify its name 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.4: Azure Tables - datastore specific configuration

The available properties are:

- `table.emulator` [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` default is **localhost**. If storage emulator is used set the value for the emulator proxy.
- `table.protocol` [http|https] default is **https**. Define the protocol to be used within requests.
- `table.partition.default` 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 configuration of 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

Configuring CPIM migration

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 the values of *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 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

ZooKeeper client must be configured in the *thread* of *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 more 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 applies the same consideration made for the thread case.

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 to IaaS).

The `<execute>` tag define if backup should or should not be performed, the possible values are *yes* or *no*. The tag is *optional* since its default is **yes**. It's supposed to be used for turning off the backup, in this case the configuration should be simply:

```
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

Configuring CPIM migration

The text in the `<prefix>` tag will be prefixed to each blob created to avoid file name conflicts. This prefix is an *optional* field since 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 file will be placed and retrieved in case they needs to be restored.

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 won't 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 required classes.

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 described the required procedure to build the benchmark project that contains the YCSB adapters. Are then shown the required commands to be executed in order to execute the benchmarks and the available values to be configured for each of the supported YCSB adapter.

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 requires to run a reachable storage emulator on Windows so if you do not want to execute test while build run `mvn clean install -DskipTests`.

Test for the Datastore extension can be executed without any configuration as they're executed thought google in-memory Datastore stub.

Run YCSB tests

Since 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`.

Now all the required dependency for kundera-benchmark should be resolved so is possible to install it with maven: `mvn clean install` 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.

C.3 Run tests for low-level API version

To run the benchmarks, after having built the project, the two phases of the YCSB tests 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. Property files must provide information to locate the database to test when running the benchmarks on the low-level API versions.

C.3 Run tests for low-level API version

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` [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` [http—https] *optional*, default is https

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

Hbase The properties must be configured inside the file 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

To run the benchmarks, after having built the project, the two phases of the YCSB tests 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.

Configuration possibilities 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`. If those values needs to be changed can be configured in the `hbase-properties.xml` file.

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].