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 italiano

Abstract

abstract in english

Table of Contents

List of Figures

Chapter 1

Introduction

Introduzione al lavoro. Inizia direttamente, senza nessuna sezione.

Argomenti trattati suddivisi sezione per sezione...

Per citare un articolo, ad esempio [4] o [1, 2] utilizzare il comando `cite`.

Per gestire i file di tipo `bib` esiste il programma `JabRef` disponibile sul sito `http://jabref.sourceforge.net/`.

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 1 ...
- In Chapter **??** ...
- In Chapter **??** ...

- …

Finally, in Chapter 7, …

Chapter 2

State of the art

2.1 Introduction

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

2.2 Summary

Riassunto del capitolo

Chapter 3

Problem setting

3.1 Introduction

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

3.2 Summary

Riassunto del capitolo

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 Kundera extension developed, 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 [3] 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)

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

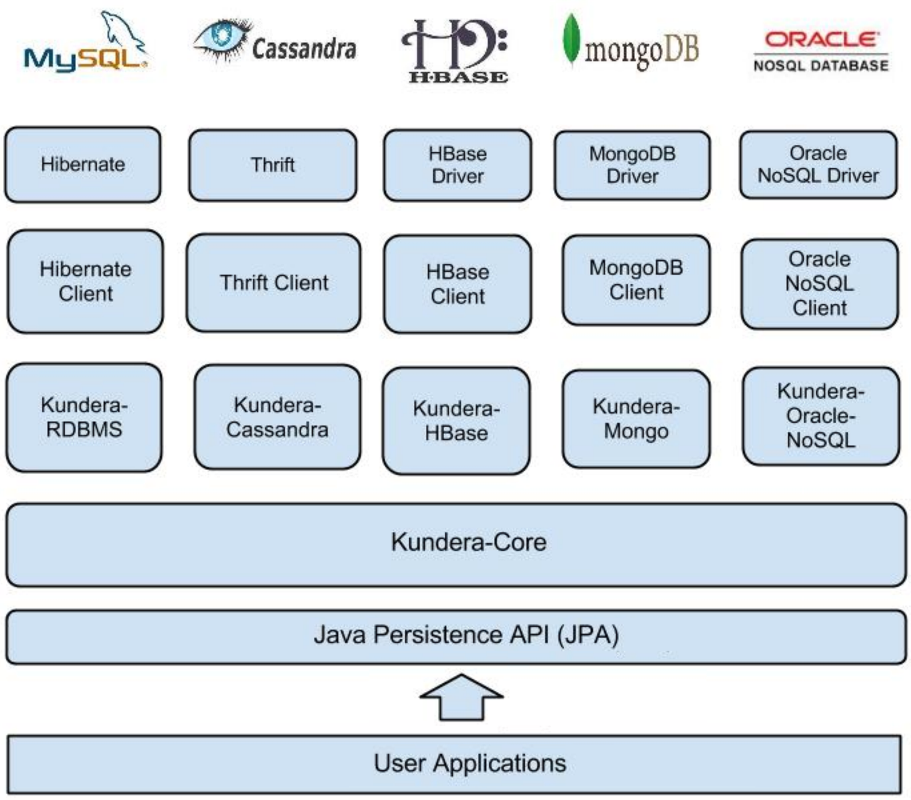

Figure 4.1: Kundera architecture

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.

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, if interested, in using and extending it adding support to other datastore. Is so available a Client Extension Framework descripted in the Kundera wiki which provides a short introduction on how Kunders clients should work and provides the 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 quite simple but the problem is that the documentation is actually outdated. Two were the main problem in understaing what to do and how, firstly it turns out that the required interfaces are actualy a little different and also are the required methods secondary, and slightly more time consuming, is that no hints nor documentation are given on the structure and informations carried by the methods arguments. The arguments carrys data structures containing informations organized in the kundera metamodel which is the implementation of the JPA metamodel that contains all the information associated (throug annotations) to a class or a field.

Due to those problems and to shrink the developing time, the solution was

to write on the Kundera google group page to ask the community for more updated infos 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 informations 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 Kundera metamodel, the most valid solution was to approach the extesion as a test driven development, so, looking at the tests code of the other clients, I've writed a set of unit tests one for each feature. With the tests failing and studyng the code of the other Kundera clients was then possible to reverse engineer the arguments thath were not documented and thus be able to develop the new extensions.

Unit tests are analyzed in detail in the chapter 6.

4.3 Developing client extensions

Have been developed two different extension for Kundera, the first one for Google Datastore and the second one for Azure Table. After a first difficulty in figuring out how the extension have to be carried out, a main structure has been defined so the two projects have many parts in common. In the following sections the extensions are presented separately, the 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 (GAE) Datastore [2] the NoSQL solution available in the App Engine runtime, is 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. Entities 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 on Datastore the idea was to use the Java class representing the datastore Key as indentifier for the POJO but unfortunately this was not possible since Kundera support only a pre defined set of Java datatypes.

The adopted solution is to handle the key internally, each time an opertion is required on Datastore the key relative to the entity is builded, the key Kind is directly mapped to the table name and the Key identifier is the user defined id 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 user specified before the persist operation but in case of ID auto-generated the field must be of type `String` and the generated ID will be a string representation of a random java `UUID`.
Auto-geenerated ID are supported by Kundera thorugh `@GeneratedValue` with `AUTO` or `TABLE` strategy, only `AUTO` strategy is supported. It was not possible to use the Datastore API to generate IDs since is necessary to know the Kind of the entity to be persisted but neither the `AUTO` strategy nor the `TABLE` one provides this infomation 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 Groups are managed in storng 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 to the user, for example in Objectify [5] the developer make use of an `@Parent` annotation that make the user able to specify the Ancestor Path. Since JPA is well defined and adding such annotation will break the standard the only alternative way is trying to automatic guess the ancestor path.

Relationsips are clearly a good point where found information for guessing if two entitiy kind can be hirearchically 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, the owning entity can be used as parent for every related entity.
- **Many to One** can be skipped since are the inverse of **One to Many** so such related entities should be already organized.
- for **Many to Many**, since the realtionship is handeled 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 [4] has been implemented in the client and have been implemented like they would be in a RDBMS system. 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* tp 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 RDBMS. For **Many to Many** relationships a join table is created based on user directives specified in the POJO 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 [4]. 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` is provided a filled metamodel of the query which needs to be *explored* in order to build a datastore compatible query. Datastore have on its own a very good support to queries so all the clauses are supported except for the *LIKE* clause.
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 [4] annotating the field that needs to be embedded with the `@Embedded` annotation and annotating the corresponding POJO with the `@Embeddable` annotation.

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

Collection fields

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

Lists are natively supported by Datastore but are supported only if composed of primitives Java datatypes. To be able to save whatever kind of collection or map independently by the datatype thath compose it, the collection or map itself is serialized to a `byte` array when persisted and deserialized when readed. 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 represention 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.
- *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 withouth restriction. Also the *create* case is usless for Datastore since if a new entity of an unknown Kind is persisted it's created without the need to explicitly define it first as a new Kind. The last case "*create_drop* will so just drop the current schema, deleting all the persisted Kind(s) 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 Datastre extension configuration see the appendix A.2.

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 AzureTable 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 `partitionKey` and a `rowKey`. 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 are supported only in field of type `String` and since the JPA annotation `@Id` can be declared on only one field per POJO, partition key adn row key are concatenated in a single `String` and hadled internally by the extension through the class `AzureTableKey` builded ad hoc since for Table ther's no a class similar to `Key` of Datastore. This way the user have complete control over partition key and row key and thus on the consistency mechanism.

Are available for the user three different approaches to handle those identifiers:

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 indipendently of the way the extension handle them, a static method `AzureTableKey.asString(String partitionKey, String rowKey)` is provided. Is not required its usage but in this case the ID 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 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 similary to RDBMS 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 and row key is not sufficient to identify an entity universally, it is sufficient for Kundera since the information of the table is available to the client just by looking at the metadata 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 deserializing 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 the supported datatypes.
To support even complex collection or maps the obvious solution is to serialize the entire collection or map to a `byte` array when persisting the entity and deserialize 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 represention 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 ther'se no fixed schema and entities within the same Table can have different properties withouth 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 creaed while still exists, put the process to sleep for an amount of time and try again
- do not delete the Table itself but delete all its entities in bulk

The first method is clearly dangerous since no deadline is given 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 a query must be performed over the table to retrieve at least partion key and row key for each entity in the table.
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 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 calls
- `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.3.

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

Table 4.1: JPQL clauses support for the developed extension

4.4 Summary

In this chpater has been introduced in details how Kundera extension should been developed, the problem encountered during the development, how tey've been addressed and the detail of the implementation of the two extensions including the feature that are currently supported. In the next chapter will be explained how has been possible to integrate Kundera into CPIM as part of the NoSQL service.

Chapter 5

CPIM extension

5.1 Introduction

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

5.2 ...

Argomenti trattati suddivisi sezione per sezione...

5.3 Figure

Per includere delle figure come la Figura 5.1 usare il comando `includegraphics`.

5.4 Algoritmi

Per includere degli algoritmi come l'Algoritmo 1 usare lo stile `algpseudocode` presente nel package `algorithmicx`.

5.5 Summary

Riassunto del capitolo

Algorithm 1 Un esempio di algoritmo.

```
Initialize Q(·,·) arbitrarily
for all episodes do
    t ← 0
    Initialize s_t
    repeat
        a_t ← π(s_t)
        perform action a_t; observe r_{t+1} and s_{t+1}
        Q(s_t, a_t) ← Q(s_t, a_t) + α(r_{t+1} + γ max_{a∈A} Q(s_{t+1}, a) − Q(s_t, a_t))
        t ← t + 1
    until s_t is terminal
end for
```

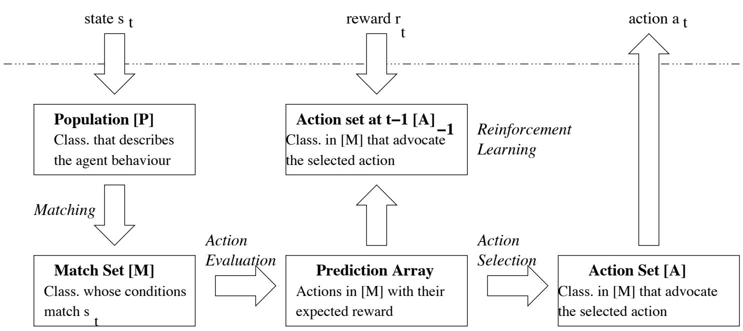

Figure 5.1: . . . titolo

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

Introduzione agli argomenti trattati nell'appendice, dalle 4 alle 10 righe.

A.2 GAE Datastore

A.3 Azure Table

A.4 . . .

Argomenti trattati suddivisi sezione per sezione. Alla fine del capitolo non includere alcun sommario.

Appendix B

Run YCSB tests

B.1 Introduction

Introduzione agli argomenti trattati nell'appendice, dalle 4 alle 10 righe.

B.2 ...

Argomenti trattati suddivisi sezione per sezione. Alla fine del capitolo non includere alcun sommario.

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

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

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