

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	1
1 Introduction	3
2 State of the art	5
2.1 Introduction	5
2.2 Summary	5
3 Problem setting	7
3.1 Introduction	7
3.2 Summary	7
4 Kundera extension	9
4.1 Introduction	9
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 extension	12
4.3.1 Google App Engine Datastore client	12
4.3.2 Queries	15
4.3.3 Azure Table client	17
4.3.4 Queries	18
4.4 Summary	20
5 CPIM extension	21
5.1 Introduction	21
5.2	21
5.3 Figure	21

TABLE OF CONTENTS

5.4	Algoritmi	21
5.5	Summary	21
6	Evaluation	25
6.1	Introduction	25
6.2	Test correctness of CRUD operations	25
6.3	Performance tests	25
6.4	Summary	25
7	Conclusions and future Works	27
	Appendices	29
A	Configuring Kundera extensions	31
A.1	Introduction	31
A.2	31
B	Run YCSB tests	33
B.1	Introduction	33
B.2	33
	Bibliography	35

List of Figures

4.1	Kundera architecture	10
5.1	...titolo	23

Chapter 1

Introduction

Introduzione al lavoro. Inizia direttamente, senza nessuna sezione.

Argomenti trattati suddivisi sezione per sezione...

Per citare un articolo, ad esempio [3] 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 ?? ...

Introduction

- ...

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. In section 4.3.1 are discussed the detail fo the extension for Google Datastore and in the section 4.3.3 the details for Azure Table.

4.2 Overview of Kundera

Kundera [3]online:kundera is an implementation for 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. In the `persistence.xml` configuration file is then possible to specify to Kundera what database is to be used for persistence. The currently supported NoSQL databases are:

- Oracle NoSQL (versions 2.0.26 and 3.0.5)
- HBase (version 0.94.3)
- MongoDB (version 2.9.1)
- Cassandra(versions1.2.9 and 2.0.4)

Kundera extension

- Redis (version 2.6.6)
- Neo4j (version 1.8.1)
- CouchDB (version 1.0.4)
- Elastic Search (version 0.90.3)

Each of the supported databases has its own client implementation that uses a factory pattern, a Client- Factory is invoked to instantiate a Client which then use the database-specific API to interact with the underlying database. The Client is invoked by Kundera-Core each time an operation has to be performed on the database.

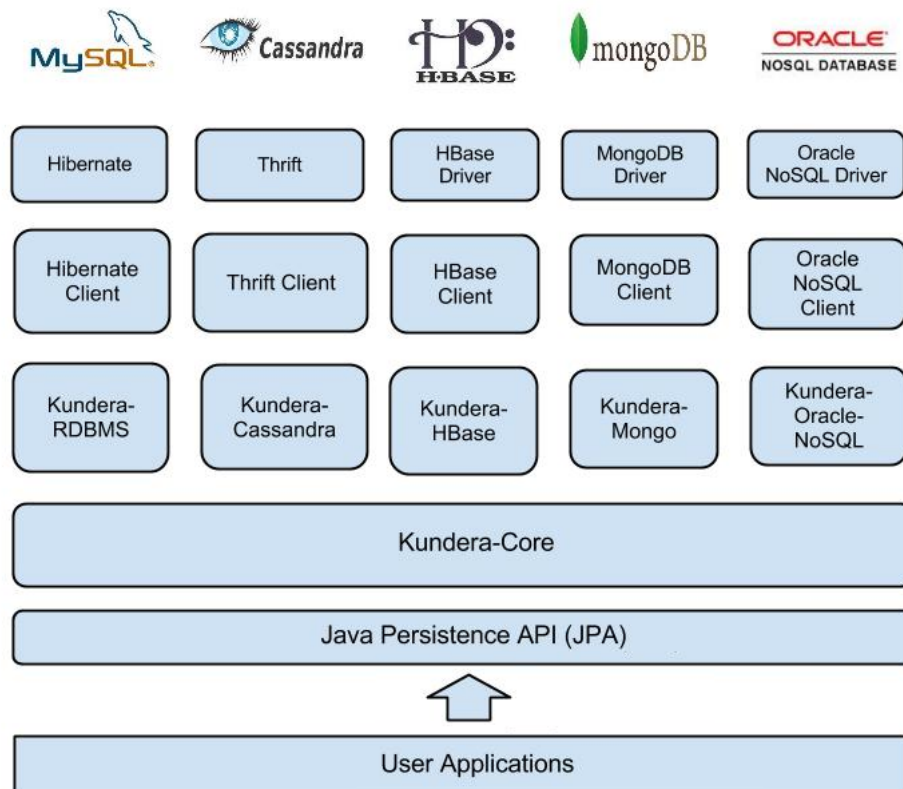


Figure 4.1: Kundera architecture

The architecture of Kundera is shown in Figure 4.1. The figure highlights the fact that the user application interacts with Kundera by exploiting the JPA interface. The Kundera-Core, in turn, interacts with the databases through the corresponding Clients.

4.2.1 Kundera's Client Extension Framework

Kundera as an open source project, thought that other developers could be interested in using it and extending its support to other datastores. So in the wiki is presented the Client Extension Framework which provides a short description on how Kundera's clients should work and provides the interfaces and classes that should be developed in order to make the client work properly.

Looking at Kundera architecture, it is clear the modularity on which Kundera has been developed. When dealing with classes JPA annotated, the Kundera core provides the necessary logic to fully support the JPA 2.1 specification and when it's time to interact with the underlying database (for persisting, updating or reading entities) it delegates the operation to the configured client in the `persistence.xml` file.

The steps to build a new Kundera client, basically 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 wiki is actually outdated. Two were the main problems in understanding what to do and how, firstly it turns out that the required interfaces are actually a little different and also are the required methods secondary, and slightly more time consuming, is that no hints are given on the structure and information carried by the methods arguments. The arguments carry data structures containing information organized in the

Kundera extension

kundera metamodel which is the implementation of the JPA metamodel that contains all the information associated (through 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 infos for the Kundera's Client Extension Framework and tell me to look forward to the other client implementation for some examples. In light of 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 extension as a test driven development, so looking at the tests code of the other clients I've written a set of unit tests one for each feature (tests are analyzed in detail in chapter 6). With the tests failing and the code of Kundera core was then possible to reverse engineer the arguments that were not documented and thus be able to develop the new extensions.

4.3 Developing client extension

Has been developed two different extensions 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 workflow has been the two projects have many parts in common. In the following sections the extensions are presented separately, the concepts are introduced the first time 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 done is the one for Google App Engine (GAE) Datastore the NoSQL solution available in the App Engine runtime, is a key-value storage build on top of Google BigTable.

JPA identifier

Google Datastore is a key-value storage in which 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 identifier for the POJO but unfortunately this is not possible since Kundera support only a well known set of Java datatypes.

The adopted solution is to handle the key internally, each time an operation is required on Datastore the key relative to the entity is build, 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 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-generated ID are supported by Kundera through @GeneratedValue with AUTO or TABLE strategy, only AUTO strategy is supported as a random Java UUID. 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 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 are in the same entity group.

In Datastore consistency is managed through Entity Groups and so by defining the ancestor paths, entities within the same Entity Groups are managed in strong consistency, eventual consistency is used otherwise.

Datastore provide the possibility to create Ancestor path by defining them parent to other entities and is basically a task leaved 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 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.

Relationships are clearly a good indicator when trying to guess if two entities kind can be hierarchically related:

- One to Many, since there's a "many" side which is the non-owning side of the relationship the owning side can be clearly used as parent for every entity in the "many" side;
- Many to One, this is the inverse of the previous type and thus entities should be already organized;
- One to One, can be treated like the One to Many
- Many to Many, in this case since there's a join table between the entities there are several solutions:
 - put the join table and the non-owning entities parent to the owning ones;
 - put all the join table, the entities on the owning side and the ones on the non-owning side under a common fictitious root entity kind

This unfortunately is not convenient since there's a lot of possibilities, think for the Many to Many case but more important is that if an entity is in more than one of those relationships it is not possible to prioritize them and choose

unless asking to the user which is the case, furthermore when declaring a entity parent to another is always necessary to know the Key of the parent beside the Key of the entity itself to be able to retrieve it from Datastore and as Kundera is structured this kind of information is not available in the client but must be serached inside the kundera metadata when possible.

For those reasons was not possible without causing strange behaviours, automatically guess the Ancestor Paths through JPA relationships so at the end is not possible for the user to manage entity consistency, 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 has been implemented like they would be in a RDBMS system. So for One to One and One to Many relationships, where on the onwer side of the relationships there's a link to the non-owning side, the connection is kepted persisting within the entity the Key (Kind and identified) of the related entity.

For the Many to One relationships there would be to solutions:

- persist a list of Key of the related entities;
- do not persist anything within the entity but fill the relationship with a query.

The second solution has been adopted since more consistent with the other client implementation and with the classic implementation of the relation type for RDBMS.

For the Many to Many relationships a join table is created based on the directives of the user specified in the annotations, then is filled each time one entity is persisted and is related with another one through a many to many relationship.

4.3.2 Queries

Here there's a comparative table for the JPQL operator supported:

Kundera extension

JPA-QL Clause	Datastore support
<i>SELECT</i>	yes
<i>UPDATE</i>	yes
<i>DELETE</i>	yes
<i>ORDER BY</i>	yes
<i>AND</i>	yes
<i>OR</i>	yes
<i>BETWEEN</i>	yes
<i>LIKE</i>	no
<i>IN</i>	yes
=	yes
>	yes
<	yes
>=	yes
<=	yes

Embedded entities

For JPA embedded entities eh implementation for Datastore is straightforward because it supports natively embedded entities.

The implementation so make use of this feature translating the embedded POJO in a Datastore embedded Entity and then persist it within the parent entity.

Collection fields

Collections and maps 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 that compose it, the collection or map itself is serialized when persisted and deserialized when readed.

Serialization and deserialization is completely handled into the extension making it completely transparent to the user.

Enumeration fields

Enum fields are supported simply by persisting its string representation and instantiating the enum type back when the entity is read.

Schema Manager

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

- `validate` which validates schema tables based on entity definition.
- `update` which updates schema tables based on entity definition.
- `create` which creates schema tables based on entity definitions.
- `create_drop` which drops (if exists) schema, then creates schema 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 useless 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 "table". The remaining case "create_drop" will so just drop the current schema deleting all the entities of all the Kinds without recreating schema since it constructs by itself.

4.3.3 Azure Table client

Table 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 is identified by the pair partition key and row key

Consistency

In Azure Table strong consistency is guaranteed while entities are stored within the same partition key otherwise consistency will be eventual. IDs are supported only in field of type String (so only a String field can be annotated with `@Id`). User can define IDs both with or without partition key.

Kundera extension

Define both row key and partition key

This can be done in two ways:

- using `AzureTableKey.asString` method by passing both partition key and row key to obtain a string representation of the whole key and assign it to the entity ID field before persist.
- manually define the entity ID before persist the entity, the string must follow the pattern `partitionKey_rowKey`.

Define only the row key

If only the row key is defined, the partition key is implicitly the default one (which can be set in a datastore specific properties file).

There are three ways to do this:

- auto-generated IDs (the row key is a random java UUID)
- manually define the entity ID before persist the entity
- using `AzureTableKey.asString` passing as parameter the desired row key and assign its result to the entity ID field before persist.

JPA relationships

Also for Azure Table, to keep uniform the extension behaviour, all the JPA supported relationships has been implemented in the client has been implemented like they would be in a RDBMS system.

The only difference is that when is needed to keep a reference to another entity in the owning side of a relationship is persisted within the entity the partition key and the row key of the related entity since the pair partition key and row key universally identify an entity.

4.3.4 Queries

Here there's a comparative table for the JPQL operator supported:

4.3 Developing client extension

JPA-QL Clause	Table support
<i>SELECT</i>	yes
<i>UPDATE</i>	yes
<i>DELETE</i>	yes
<i>ORDER BY</i>	no
<i>AND</i>	yes
<i>OR</i>	yes
<i>BETWEEN</i>	yes
<i>LIKE</i>	no
<i>IN</i>	no
=	yes
>	yes
<	yes
>=	yes
<=	yes

Embedded entities

Embedded fields are not supported natively by Azure Table so the solution adopted is to serialize the field annotated as embedded to be able to save it to the storage and deserializing it when the entity is read.

Collection fields

Collection and maps are not supported by Azure Table since it supports only a set of primitive Java datatypes. To support even complex collection or maps the solution is, like Datastore, serialize the entire collection or map when persisting the entity and deserializing it when reading the entity from the storage.

Enumeration fields

Enum fields are supported simply by persisting its string representation and instantiating the enum type back when the entity is read.

Schema Manager

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

- validate which validates schema tables based on entity definition.
- update which updates schema tables based on entity definition.
- create which schema tables based on entity definitions.
- create_drop which drops (if exists) schema, then creates schema tables based on entity definitions.

Here, like Google Datastore, the first two cases are quite useless for Azure Table since there's no fixed schema and entities within the same Table can have different properties without restriction.

Azure Table needs that the Table in which entities are stored exists before trying to create entities so the "create" case simply iterates over all table names and creates it in the database. For the "create_drop" case, all tables are dropped (and so all the contained entities) and re-created.

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 what the feature 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.

```
1: Initialize  $Q(\cdot, \cdot)$  arbitrarily
2: for all episodes do
3:    $t \leftarrow 0$ 
4:   Initialize  $s_t$ 
5:   repeat
6:      $a_t \leftarrow \pi(s_t)$ 
7:     perform action  $a_t$ ; observe  $r_{t+1}$  and  $s_{t+1}$ 
8:      $Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha(r_{t+1} + \gamma \max_{a \in A} Q(s_{t+1}, a) - Q(s_t, a_t))$ 
9:      $t \leftarrow t + 1$ 
10:  until  $s_t$  is terminal
11: end for
```

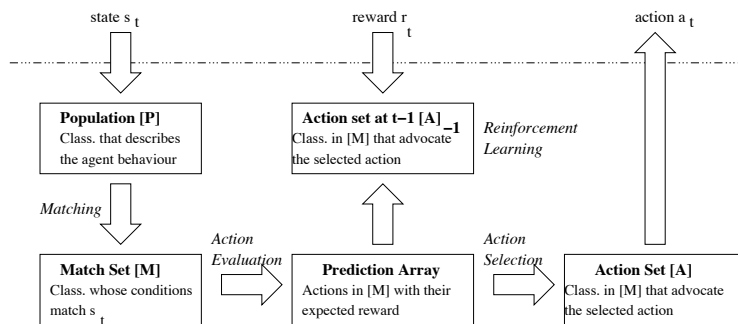


Figure 5.1: ...titolo

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 ...

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/appengine/docs/java/datastore/entities>.
[Online].
- [3] Schincariol Merrick Keith Mike. *Pro JPA 2*. Apress, Berkely, CA, USA, 2nd edition, 2013.