ONDM: an Object-NoSQL Datastore Mapper

Luca Cabibbo
Dipartimento di Ingegneria
Università Roma Tre
Rome, Italy
cabibbo@dia.uniroma3.it

ABSTRACT

ONDM (*Object-NoSQL Datastore Mapper*) is a framework to facilitate the storage and retrieval of persistent objects in NoSQL datastore systems. Recently, several NoSQL datastore systems have been implemented, each with a different data model and API to access the data. ONDM aims at supporting several challenges posed to application developers by this heterogeneity. It provides developers with a uniform application programming interface, transparent access to different NoSQL datastores, and the ability to select from different data representation techniques for entity objects and relationships between objects.

1. INTRODUCTION

The last decade has witnessed the implementation of a new generation of non-relational databases, often called NoSQL datastore systems [2, 3]. These systems support the design and development of applications that require managing persistent data, but for which traditional RDBMS's are not well suited. A common case are Web 2.0 and "social" applications — which require good horizontal scalability for a database access based on simple read-write operations, to be distributed over a cluster of datastore nodes.

According to [5], more than fifty NoSQL systems have been already implemented, each with different characteristics (e.g., different data models and API's to access the data, as well as different consistency and durability guarantees). This heterogeneity raises significant problems and challenges to application developers [5]. For example: the choice of the datastore best suited to the needs of an application under development; the organization of data in the selected datastore; a proper understanding and usage of the data access operations offered by the datastore; the need to overcome the lack of an efficient "join operation" in NoSQL systems, and the consequent need to store data in a redundant, denormalized form. Often, a lot of "try and see" experiments are required to identify a suitable design solution.

With the goal of facing up these challenges, we are developing *ONDM* (*Object-NoSQL Datastore Mapper*), a framework to facilitate the storage and retrieval of persistent objects in NoSQL datastore systems.

The approach pursued by ONDM is based on the observation that, fortunately, the various NoSQL systems share a number of goals, and common characteristics can be identified among them. Cattell [2] classifies them in a few main datastore categories, such as key-value stores (e.g., Oracle NoSQL[1] and Redis[2]), document stores (e.g., MongoDB[3] and Couchbase[4]), and extensible record stores (e.g., Apache Cassandra[5]). As suggested by [1], such commonalities can be exploited to define a uniform application programming interface to access, in a transparent way, different NoSQL datastores.

Another important trend that influenced the design and development of ONDM is the affirmation, in recent years, of ORM (Object-Relational Mapping) frameworks to increase developers' productivity in implementing data access. However, as the name suggests, ORM frameworks are tailored to relational DBMS's. Some NoSQL systems provide access to its specific datastores in an ORM-like style (e.g., Ohm for Redis[6]), and some ORM frameworks support, usually as a limited extension, a few NoSQL datastores (e.g., Hibernate Object/Grid Mapper[7]). However, in our opinion, the currently available frameworks are insufficient to fully cope with all the challenges described above.

The highlights of ONDM are as follows:

- ONDM offers to application developers an ORM-like API, a variant of the popular Java Persistence API (JPA [4]). By adopting this standard, it is easy to move existing JPA applications into the NoSQL realm, and for developers it is simple to start writing NoSQL-based applications.
- As JPA, the ONDM data model is based on entities, relationships, and embeddable objects (i.e., complex values).
- ONDM currently supports the access to a handful of NoSQL datastores (Apache Cassandra, Couchbase, MongoDB, Oracle NoSQL, and Redis), belonging to different datastore categories. More important, it has been designed to be easily extensible. Indeed, the effort needed to implement in ONDM the access to another datastore is rather limited.

[1] `http://www.oracle.com/technetwork/products/nosqldb`
[2] `http://redis.io`
[3] `http://www.mongodb.org`
[4] `http://www.couchbase.com`
[5] `http://cassandra.apache.org`
[6] `http://ohm.keyvalue.org`
[7] `http://www.hibernate.org/subprojects/ogm.html`

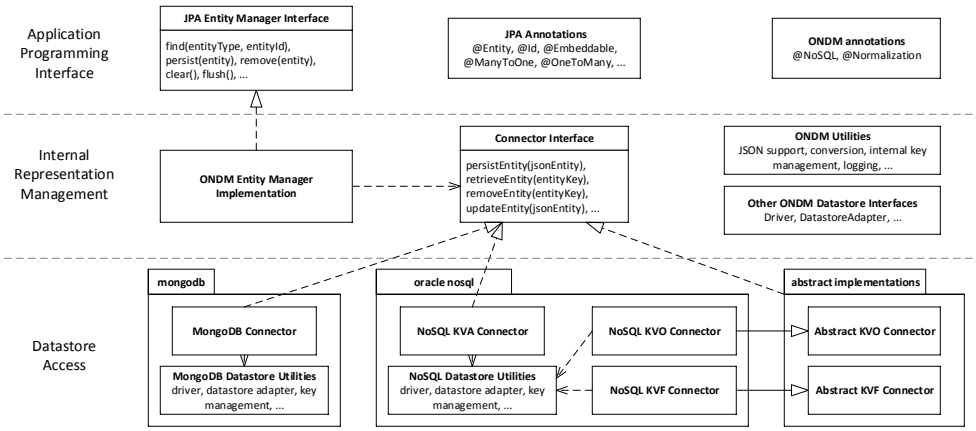

Figure 1: Abridged system architecture

- For each NoSQL datastore, ONDM can implement multiple entity representation strategies. For example, consider a key-value store, which organizes its data as a collection of key-value pairs. Representation strategies to persist an entity instance in a key-value store include: storing the entity using a single key-value pair; or storing the entity using multiple key-value pairs, one for each field of the entity. Moreover, ONDM has been designed to easily incorporate new entity representation strategies.
- Relationships between entities can be represented either as references or by materializing the associated entities.

In this demonstration we will show how ONDM addresses the challenges that application programmers should face up to use NoSQL datastores. In particular, ONDM allows programmers to experiment with NoSQL datastores in a flexible way — and without requiring any change in the application code. The flexibility we will demonstrate is based on the following options offered by ONDM:

- Independence from the specific datastore used; it is possible to move a JPA-based application from a relational database to a NoSQL datastore and then to other ones.
- Different entities (of a same application) can be stored using distinct representation strategies, or even in distinct NoSQL datastores (polyglot NoSQL persistence [3]).
- Each relationship can be stored using a distinct relationship representation mode (reference or materialization).

In the following, we briefly present the architecture of ONDM (Section 2) and provide some details on the data representations currently implemented by ONDM (Section 3). Then, we present a detailed demonstration scenario (Section 4).

2. ARCHITECTURE

ONDM is composed of three main layers (see Figure 1).

The *application programming interface* (API) consists of: (i) annotations, to describe the structure of persistent objects (called entities) and the mapping to the underlying datastores; and (ii) the entity manager, an interface offering CRUD operations to manipulate entities. The API offered to developers is based on JPA, with a few extensions.

A second layer is responsible of the *internal management of entities* within the current unit of work (i.e., transaction). This is based on an internal representation of entities, which is independent of any specific physical datastore representation. On the other hand, the responsibility to read/write these entity representations from/to specific datastores is given to connectors (described next).

Datastore access is carried out by *connectors*, each of which is tailored to a specific datastore. ONDM can provide multiple connectors for a same NoSQL system — each implementing a different entity representation strategy for that datastore. Indeed, each connector is also responsible for the conversions between the common internal representation and a specific entity representation strategy.

These main elements of this architecture will be described in more detail in the remainder of the section.

Application Programming Interface. ONDM provides an API that is largely based on JPA [4], which allows programmers to define the mapping between the persistent classes and the underlying datastores. JPA adopts a conceptual entity model — based on entities, relationships, and embeddable objects — whose constructs are associated to programming elements (classes and fields) by means of annotations (i.e., keywords prefixed by `@`). Annotation `@Entity` specifies that a class is an *entity type*, that is, a persistent class whose instances (called *entities*) have independent existence and are identified by means of a persistent *identifier* (a field annotated by `@Id`). `@Embeddable` specifies a persistent class without independent existence (its objects should be embedded in entities to be persisted). Fields representing relationships are annotated in a way depending on the relationship multiplicity, e.g., `@ManyToOne`.

ONDM introduces two additional annotations: `@NoSql` states that an entity type should be persisted in a NoSQL datastore (the annotation can include directives for a specific datastore and entity representation strategy). `@Materialized` specifies that a relationship should be represented by materializing the associated entities (in a denormalized mode) rather than by means of references, which is the default.

As running example, consider entity types `Player` and `Game`, shown in Figure 2. According to the mapping (see the `@NoSql` annotations), they will be persisted in Oracle NoSQL, but using two distinct entity representation strategies. Moreover, relationship `games` in `Player` will be stored using a materialized (i.e., denormalized) representation. In-

```
@Entity @NoSql(datastore="OracleNoSQL", strategy="KVF")
class Player {
    @Id String userName;
    @OneToMany @Materialized List<Game> games;
}

@Entity @NoSql(datastore="OracleNoSQL", strategy="KVE")
class Game {
    @Id long id;
    @ManyToOne Player firstPlayer;
    @ManyToOne Player secondPlayer;
    @Embedded GameDetails gameDetails;
}
```

Figure 2: Sample entity types

tuitively, this means that the representation of a player should include the games he is playing. The use of a materialized relationship should guarantee that, whenever we access a player in the datastore, we will also load all his games in an efficient way. `GameDetails` is an embeddable class, whose details have been omitted due to space limitations.

JPA also defines an *entity manager* interface, used by developers to execute CRUD (Create, Read, Update, Delete) operations on entities. This interface is implemented by the manager of the internal representation, described next.

Internal Representation. The main responsibility of this element is the conversion between in-memory entity objects and their internal representation. Specifically, ONDM adopts an internal representation whose syntax is based on the popular data-interchange format JSON. When the application needs to save an entity, this is first converted into the internal representation, and then persisted in a datastore using a suitable connector. On the other direction, when the application needs to retrieve an entity, its internal representation is first loaded from a datastore using a connector, and then it is converted into a plain object. The internal representation takes care of some aspects of the object-datastore mapping, (such as the relationship representation modes), but it is independent of any specific physical datastore representation.

We would like to point out that the ONDM internal representation is not a simple object serialization in JSON. (Serialization is the process of translating an object or a set of objects into a binary or textual format that can be stored and then reconstituted later.) In our internal representation, each entity instance has an individual and independent representation. The representation of an embeddable element is nested in the representation of the entity instance owning the element. Moreover, a relationship linking an entity e_1 to another entity e_2 is stored according to the representation mode specified in the mapping: if the relationship is materialized, then the representation for e_1 contains the representation of the value of e_2; otherwise, the representation for e_1 contains just a reference to e_2. The internal representation takes care also of loops through materialized relationships (which need to be splitted).

For example, Figure 3 shows the internal representation for a pair of players and a game between them, according to the entity types and mapping specified in Figure 2.

Datastore Access. The main responsibility of the datastore access layer is the conversion between the ONDM internal representation and physical datastore representations, implemented by connectors. Each *connector* implements data access for a specific datastore, and it is therefore tailored to the specific data model and API offered by that datastore.

```
Player:Luca
{
    "userName" : "Luca",
    "games" : [ { "id" : "2421", "firstPlayer" : "Player:Luca",
                  "secondPlayer" : "Player:Francesca", "gameDetails" : ... }, ... ]
}

Player:Francesca
{
    "userName" : "Francesca",
    "games" : [ { "id" : "2421", "firstPlayer" : "Player:Luca",
                  "secondPlayer" : "Player:Francesca", "gameDetails" : ... }, ... ]
}

Game:2421
{
    "id" : "2421", "firstPlayer" : "Player:Luca",
    "secondPlayer" : "Player:Francesca", "gameDetails" : ...
}
```

Figure 3: Sample entities (internal representation)

It is worth noting that, for each specific NoSQL datastore, it is often the case the same information can be persisted there in multiple ways. Indeed, each connector implements a particular way of representing the data (entity representation strategy) in a specific datastore system. In practice, ONDM can (and does) implement multiple connectors for each NoSQL system. Entity representation strategies and their implementation in connectors are discussed in Section 3.

Another responsibility of this element involves the selection of the connector to be used to perform a specific datastore access. For example, when the internal representation of an entity should be persisted in a datastore. This selection is driven by the object-datastore mapping, using the directives specified in the `@NoSql` annotation.

3. DATA REPRESENTATION STRATEGIES AND CONNECTORS

An *entity representation strategy* specifies how the internal representation of an entity instance can be stored in a NoSQL database. To this end, we have first identified a number of general representation strategies that are independent of any NoSQL datastore. Moreover, we have provided the implementation of these strategies with respect to specific datastores.

We consider an abstract data model for NoSQL datastores, which organizes data as a collection of key-value pairs. Then, each general representation strategy specifies which key-value pairs should be used to store a certain entity instance. Furthermore, each connector arranges how to store each key-value pair in a specific datastore.

A simple strategy, called KVE (*Key-Value per Entity*), consists in storing an entity instance using a single key-value pair. The key is a reference to the entity instance (e.g., `Player:Luca`) and the value is its internal representation (e.g., {`"userName":"Luca", ...`}). See Figure 4(a).

Another strategy, called KVF (*Key-Value per Field*), consists in storing an entity instance using multiple key-value pairs, one for each top-level field of the entity. For each field of the entity, the key is a reference to the entity instance plus the name of the field, and the value is the value of the specific field. See Figure 4(b).

A further strategy, called KVA (*Key-Value per Atomic value*), consists in storing an entity instance using multiple key-value pairs, one for each atomic value in the representation of the entity. For each atomic value of the entity, the key is a reference to the entity instance plus the path to that specific value. See Figure 4(c).

key	*value*
Player:Luca	{"userName":"Luca", ...}

(a) KVE

key	*value*
Player:Luca/userName	Luca
Player:Luca/games	[{"id":"2421", "firstPlayer":"Player:Luca", ...}, ...]

(b) KVF

key	*value*
Player:Luca/userName	Luca
Player:Luca/games/0/id	2421
Player:Luca/games/0/firstPlayer	Player:Luca
Player:Luca/games/0/secondPlayer	Player:Francesca
Player:Luca/games/0/gameDetails	...
...	...

(c) KVA

Figure 4: Entity representation strategies

Let us now consider *relationship representation modes*. Figure 5 illustrates their impact with reference to relationship field `games` in entity type `Player`. (The figure is based on the KVF entity representation strategy.) If the relationship is materialized, then the representation of a `Player` aggregates, in a denormalized form, the representations of his `games`. See Figure 5(a). But if the relationship is normalized, then the `Player` contains just the references to the `games` he is playing. See Figure 5(b).

We now discuss connectors. Each connector implements a representation strategy for a datastore — by mapping the key-value pairs of the abstract data model described above to the data model and API offered by a specific datastore.

A connector for MongoDB (a document store) can be based on the KVE strategy, storing each entity instance as a distinct document. In MongoDB, each document is identified by a collection and an id. These identifiers can be obtained from the key (of our abstract data model) associated with the entity instance.

The KVE strategy has been applied in a connector for Redis, a key-value store. However, since Redis offers a number of advanced features (notably, hashes and efficient multi-read operations), we have also implemented an efficient connector for Redis based on the KVF strategy.

Oracle NoSQL is another key-value store, offering structured keys and some efficient multi-read operations. ONDM provides connectors implementing the three strategies above (KVE, KVF, and KVA) for Oracle NoSQL.

Apache Cassandra is an extensible record store, where data is organized in tables (column families), rows, and columns. An entity instance can be stored as a single row of a table, comprising several columns, one for each field of the entity. This representation essentially corresponds to our strategy KVF.

Thus, a main responsibility of an ONDM connector is managing the mapping between the keys and values of our abstract data model and the specific constructs available in the datastore. Furthermore, each connector should exploit the data access operations offered by the datastore to provide efficient read-write operations of entity instances. According to our experience, the use of specific data access operations can improve performances by a factor from 1.5 to 5 (over the use of basic read-write operations).

Despite the differences in models and API's offered by the various NoSQL systems, there are also commonalities, which we have exploited in ONDM. In general, the implementation of a new connector *from scratch* requires the realization of about twenty operations (and over a thousand lines of code). However, we have also defined a few abstract connector implementations for the data representation strategies outlined above. Using such abstract implementations, the realization of a new connector requires only two hundred lines of code.

key	*value*
Player:Luca/userName	Luca
Player:Luca/games	[{"id":"2421", "firstPlayer":"Player:Luca", ...}, ...]

(a) Relationship `games` is materialized

key	*value*
Player:Luca/userName	Luca
Player:Luca/games	[Game:2421, ...]

(b) Relationship `games` as a reference

Figure 5: Relationship representation modes

4. DEMONSTRATION

The demonstration will show how ONDM allows an application programmer to experiment with NoSQL datastores in an easy and flexible way. The demonstration will be based on the following scenarios.

In a first scenario, we will demonstrate how a JPA-based application running over a relational database can be ported into the NoSQL realm. To do so, it is just required to add the `@NoSQL` annotation to each persistent class (entity) of the application. The entities will be stored in a default NoSQL datastore, according to an ONDM configuration file.

In a second scenario, we will show how to select a specific datastore and entity representation strategy for each entity type (Figure 2). Using a visual browser, we will show how sample data are represented differently in the various systems and according to diverse data representation strategies (Figure 4). By examining application logs, we will also show how generic CRUD operations (used at the application level) are mapped to specific read/write operations of each NoSQL system. These logs also reveal the internal data representation (Figure 3), which is used to move data between the internal representation layer and the datastore access layer.

In a third scenario we will demonstrate the flexible management of relationships in ONDM. In particular, we will show the impact of changing the representation mode for relationships (using annotation `@Materialized`) on the internal representation and then on the specific physical representation in the various NoSQL datastores (Figure 5).

In a further scenario, we will consider various application workloads (write only, read only, read and write, read and update). We will measure the running times of these workloads under different choices for the datastores, the entity representation strategies, and the relationship representation modes, to investigate their impact on performances.

In doing so, we will not change any line of code of the application — apart from the annotations associated to persistent classes.

5. REFERENCES

[1] P. Atzeni, F. Bugiotti, and L. Rossi. Uniform access to non-relational database systems: The SOS platform. In *CAiSE 2012*, pages 160–174, 2012.

[2] R. Cattell. Scalable SQL and NoSQL data stores. *SIGMOD Record*, 39(4):12–27, 2010.

[3] M. J. Fowler and P. J. Sadalage. *NoSQL Distilled*. Addison-Wesley, 2012.

[4] J. P. . E. Group. The Java Persistence API 2.0, jsr 317, 2009.

[5] M. Stonebraker. Stonebraker on NoSQL and enterprises. *Commun. ACM*, 54(8):10–11, 2011.