An Object-Datastore Mapper Supporting NoSQL Database Design

Francesca Bugiotti
Università Roma Tre, Italy
bugiotti@dia.uniroma3.it

Luca Cabibbo
Università Roma Tre, Italy
cabibbo@dia.uniroma3.it

ABSTRACT

Recently, several NoSQL database systems have been developed, each with different data models and APIs to access the data. This heterogeneity poses several challenges to data and application designers. The same application data can be organized in a datastore according to multiple different data representations. Significant decisions are then required, since data organization impacts important qualities, such as scalability and performance. Application developers and data designers can therefore benefit from tools assisting them in performing experiments to choose among various alternative data representations, in a same or in different NoSQL systems.

In this paper we propose ONDM (Object-NoSQL Datastore Mapper), a framework that supports the management of persistent objects in NoSQL datastores. It provides application developers with a uniform programming interface, as well as the ability to map application data to different data representations, in a flexible way, towards a variety of NoSQL systems. In virtue of these characteristics, ONDM can also be used to support, in an effective way, developers in performing the experiments needed during the design of a NoSQL database.

1. INTRODUCTION

The last decade has witnessed the implementation of a new generation of non-relational databases, often called NoSQL datastores [8, 19]. These systems support the development of applications that require the management of persistent data, but for which traditional relational DBMSs are not well suited. A common case are next-generation web applications (i.e., Web 2.0 and "social" applications) — simple OLTP-like applications, which require a data access based on simple read-write operations, together with good horizontal scalability and performance.

Overall, in this paper we are concerned with the problem of mapping the persistent data of a modern web application to a NoSQL database. Even if NoSQL datastores are often described as "schemaless," the data of interest for applications do usually have some structure, that should be mapped to the modeling elements (collections, tables, documents, key-value pairs) available in the target system. This problem is particularly challenging, since the organization of data in the datastore affects many quality properties (including scalability, performance, and consistency), and thus it requires significant design decisions [5, 7].

In general, there are different alternative data representations for the same data (and possibly many for each candidate target datastore). A NoSQL database design methodology can help in identifying the possible valid representations, but, as of today, they are not yet mature enough to help selecting, in a speculative way, the data representation that better fits a given usage scenario. Rather, a lot of "try and see" experiments are then usually needed to assess the most suitable design solution [19]. For example, a benchmark to measure the relative performances of important operations under the expected workload. In general, these experiments need to be repeated for different target systems and data representations. Unfortunately, implementing the same data access operations over multiple systems and data representations is time consuming and error prone. In this context, we believe that application and data designers would benefit from tools that allow mapping the data of an application to different data representations, over the data structures of various target NoSQL systems. Such tools would assist developers in coping with data design decisions that require performing experiments in a flexible way.

To this end, we propose *ONDM* (*Object-NoSQL Datastore Mapper*), a framework that provides application developers with a uniform access towards a variety of NoSQL systems, and the ability to map application data to different data representations, in a flexible way. In virtue of these characteristics, ONDM supports application developers and database designers in performing the experiments needed during the design of a NoSQL database.

The main features of ONDM are as follows:

- ONDM provides application developers with an object-oriented API which is a variant of the popular Java Persistence API (JPA [10]). As such, its data model is based on entities, value objects, relationships, and aggregates. 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.
- ONDM implements the transparent access to various NoSQL datastores. The specific system to be used by

an application can be specified in a declarative manner. Therefore, this choice can be easily modified, wit limited or no impact in the source code.

- ONDM is able to handle data representations in a flexible way. Specifically, they can be specified in a declarative manner, in terms of a language to express custom data representations [7]. This choice can be easily modified, too.

- In general, ONDM is based on the abstract data model and language to specify data representations introduced in NoAM [7], a system-independent approach for NoSQL database design.

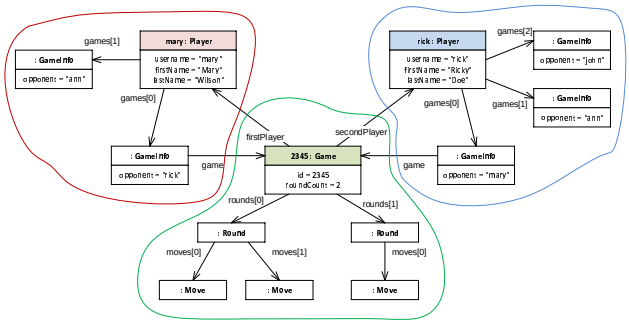
The paper is organized as follows: Section 2 provides an overview of the issues in NoSQL database design, and of how ONDM allows to cope with them. Section 3 introduces the architecture of ONDM, and then its main components are described in more details in Sections 4, 5, 7, and 8. Section 6 summarizes relevant ideas from NoAM [7], including the abstract data model and the language for custom data representations, on which ONDM is based. Section 9 presents a case study in the usage of ONDM. Section 10 discusses related work. Finally, Section 11 draws some conclusion.

2. CONTEXT AND OVERVIEW

With respect to the problem of mapping the persistent data of an application to a NoSQL datastore, we will now discuss the challenges it raises, and how the ONDM framework can be used to deal with them. To this end, we consider, as a running example, a next-generation web application: a fictitious online game. This is indeed a typical NoSQL usage scenario. In general, in this paper we mainly refer to "simple" OLTP-like applications [8]. We do not consider here other classes of applications, such as those performing complex processing of large datasets.

We assume here that the design of our application follows Domain-Driven Design (DDD) [9], a popular object-oriented design methodology. First, persistent objects are classified in entities and value objects. An *entity* is a persistent object that has independent existence and is distinguished by a unique *identifier*. A *value object* is a persistent object which is mainly characterized by its value, without an own identifier. Then, we group entities and value objects into aggregates, with boundaries around each. Each *aggregate* has an entity as its root, and can also contain many value objects. Intuitively, an entity and a group of value objects are used to define an aggregate having a complex structure and value. Aggregates are intended to be units of data access (as operations access individual aggregate objects), distribution (to support scalability), and consistency (business invariants can be defined over the boundary of single aggregate objects). The design of aggregates, that is, the identification of the various classes of aggregate objects needed in an application, is driven by use cases (functional requirements). This activity is described, for example, in [9, 12, 21].

Our game application should manage various collections of objects, to represent players, games, and rounds. Figure 1 shows a few representative application objects. In our example, players and games are aggregate objects, but rounds are not. Note also that each game (as an aggregate) contains (by



Legenda: Boxes and arrows denote *objects* and *links* between them, respectively. An object having a colored top compartment is an *entity*, otherwise it is a *value object*. A closed curve denotes the boundary of an *aggregate*.

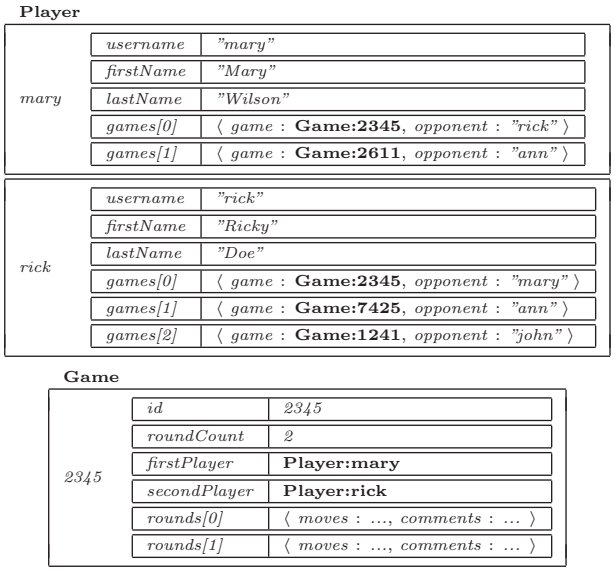
Figure 1: Sample application objects

Player:mary : ⟨
  username : *"mary"*,
  firstName : *"Mary"*,
  lastName : *"Wilson"*,
  games : {
    ⟨ *game* : **Game:2345**, *opponent* : *"rick"* ⟩,
    ⟨ *game* : **Game:2611**, *opponent* : *"ann"* ⟩
  }
⟩
Player:rick : ⟨
  username : *"rick"*,
  firstName : *"Ricky"*,
  lastName : *"Doe"*,
  games : {
    ⟨ *game* : **Game:2345**, *opponent* : *"mary"* ⟩,
    ⟨ *game* : **Game:7425**, *opponent* : *"ann"* ⟩,
    ⟨ *game* : **Game:1241**, *opponent* : *"john"* ⟩
  }
⟩
Game:2345 : ⟨
  id : *"2345"*,
  roundCount : *2*,
  firstPlayer : **Player:mary**,
  secondPlayer : **Player:rick**,
  rounds : {
    ⟨ *moves* : ..., *comments* : ... ⟩,
    ⟨ *moves* : ..., *comments* : ... ⟩
  }
⟩

Figure 2: Aggregate objects as complex values

nesting) the rounds played by the opponent players. Moreover, Fig. 2 shows a representation of the aggregates of Fig. 1 as complex values.

Let us now consider how application data can be represented in a NoSQL datastore. In this paper, we refer to the NoAM abstract data model for NoSQL databases [7], which is intended to generalize the common aspects of the specific modeling features of NoSQL datastores, in a system-independent way. In NoAM, data are organized in *collections*, *blocks*, and *entries*. With reference to major NoSQL categories [8] we have that, in document stores (e.g., MongoDB [16]), collections, blocks, and entries correspond to document collections, documents, and document fields, respectively. In extensible record stores (e.g., Amazon DynamoDB [2]), they correspond to tables, rows, and columns, respectively. In key-value stores (e.g., Oracle NoSQL [17]), entries and blocks correspond to key-value pairs and groups of related key-value pairs, respectively.



Legenda: Inner boxes show *entries*, while outer boxes denote *blocks*. *Collections* are shown as groups of blocks

Figure 3: A possible candidate data representation for the sample aggregate objects of Fig. 1 and 2

NoSQL data design deals with the mapping of application data elements (such as entities, value objects, aggregates, and their values) to a NoSQL database. For example, the aggregate objects of Fig. 1 and 2 can be represented in the NoAM data model as shown in Fig. 3.

Furthermore, the NoAM approach provides implementations for its abstract data model in various NoSQL datastores. For example, Fig. 4 shows how the abstract database of Fig. 3 can be represented in the Oracle NoSQL key-value store [17].

In general, application data elements can be mapped to a NoSQL database in different alternative ways. For example, Fig. 5 and 6 show two further (and both valid) representations for the game aggregate object of Fig. 1 and 2.

These candidate representations could have been identified to support different application use cases. For example, the data representation of Fig. 5, which uses a single entry to represent a whole game, supports the efficient retrieval of a game (when a player selects to continue it), on the basis of the guideline: "If all or most of the data of an aggregate are accessed or modified together, then it should be represented by a single entry." On the other hand, the data representations of Fig. 3 and 6, which split different rounds in distinct entries, support the efficient addition of a round to a game (when a player completes a round in a game she is playing), on the basis of the guideline: "An aggregate should be partitioned in multiple entries if there are operations that frequently access or modify only specific portions of the aggregate." (The guidelines are adapted from [7] and [6].)

As of today, a NoSQL database design methodology can help in identifying the valid data representations for an application dataset but, then, a lot of "try and see" experiments are usually needed to assess the data representation that better fits a given usage scenario [19]. For example, the choice among the above data representations (and possibly

key	
/major/key/-/minor/key	*value*
Player/mary/-/username	*"mary"*
Player/mary/-/firstName	*"Mary"*
Player/mary/-/lastName	*"Wilson"*
Player/mary/-/games[0]	{*game*: **"Game:2345"**, *opponent*: *"rick"*}
Player/mary/-/games[1]	{*game*: **"Game:2611"**, *opponent*: *"ann"*}
Player/rick/-/username	*"rick"*
Player/rick/-/firstName	*"Ricky"*
Player/rick/-/lastName	*"Doe"*
Player/rick/-/games[0]	{*game*: **"Game:2345"**, *opponent*: *"mary"*}
Player/rick/-/games[1]	{*game*: **"Game:7425"**, *opponent*: *"ann"*}
Player/rick/-/games[2]	{*game*: **"Game:1241"**, *opponent*: *"john"*}
Game/2345/-/id	*2345*
Game/2345/-/roundCount	*2*
Game/2345/-/firstPlayer	**"Player:mary"**
Game/2345/-/secondPlayer	**"Player:rick"**
Game/2345/-/rounds[0]	{*moves*: ..., *comments*: ...}
Game/2345/-/rounds[1]	{*moves*: ..., *comments*: ...}

Figure 4: Implementation in Oracle NoSQL for the data representation shown in Fig. 3

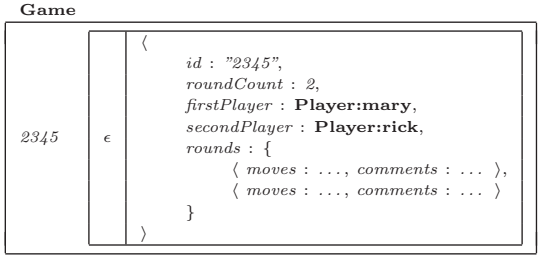


Figure 5: A second candidate representation

others) could be based on the relative performance of important operations, measured under an expected workload.

However, performing experiments on NoSQL datastores is a difficult activity, for several reasons. First, each datastore has its own data model and API to access the data, and it is often difficult to understand how to exploit the data access operations they offer. Second, the coding required to implement different data representations can be time-consuming and error prone. The former issue can be mitigated by using a framework to access different NoSQL datastores in a uniform way. The latter issue can be dealt with by using a flexible mapper framework for NoSQL datastores, that maps the data organization in the application to the data structures in the database. However, to the best of our knowledge, currently there is no tool which possesses all the above features.

In this paper, we aim at filling this gap, by presenting ONDM, a NoSQL mapper framework which allows developers to access different NoSQL datastores using a unified programming interface, and to manage different data representations in a flexible and customizable way.

In ONDM, application data are organized in objects and classes. Then, classes are annotated using declarations to specify the target system and data representation for each class. See, for an example, Fig. 7. There, the various annotations (keywords prefixed by `@`) specify metadata on the data mapping. `@Entity` means that `Game` objects are persistent objects. Annotation `@NoSql` is used to select a target datastore (Oracle NoSQL, in this case). Annotation `@DataRepresentation` is used to specify a data representation for `Game` objects (in this case, the one shown in Fig. 6). In order

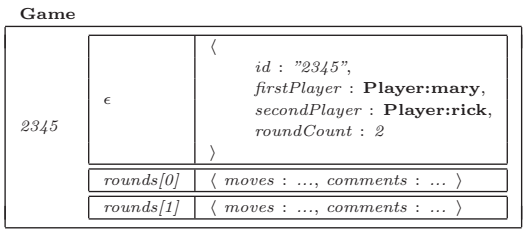


Figure 6: A third candidate representation

```
@Entity
@NoSql(datastore="OracleNoSQL")
@DataRepresentation(rules="*/rounds[*];*")
class Game {
    @Id long id;
    int roundCount;
    @ManyToOne Player firstPlayer;
    @ManyToOne Player secondPlayer;
    List<Round> rounds;
}
```

Figure 7: Sample data declarations and mapping in ONDM

to support flexibility, ONDM makes use of the language to specify data representations proposed in NoAM [7]. The framework interprets such declarations, and uses them to map objects and CRUD operations over them to data structures and data access operations for a specific NoSQL system, according to the specified data representation. Developers can select a different target system or a different data representation by just changing such declarations — and no other change in the application code is required. This way, ONDM enables the flexible prototyping and experimentation with different NoSQL systems and data representations, as required to assess a suitable data design solution.

3. ARCHITECTURE

We will now present the architecture of *ONDM* (*Object-NoSQL Datastore Mapper*), which is composed of four main layers: application programming interface, internal aggregate manager, data representation manager, and datastore adapters (see Fig. 8). These main elements of ONDM will be described in more detail in the remaining sections of the paper.

The *Application Programming Interface* (*API*) provides facilities (annotations and/or configuration metadata) to describe the structure of persistent objects (entities and value objects, their composition into aggregates, and relationships) as well as the mapping to the underlying datastores. Moreover, it defines a persistence manager, a common programming interface offering CRUD operations to manipulate entities and aggregates. The API offered to developers is mostly based on JPA [10], with a few variations.

The *Internal Aggregate Manager* (*IAM*) is responsible of the conversion between in-memory application objects (see Fig. 1) and an internal representation for aggregates (similar to that shown in Fig. 2). This internal representation is based on a complex-value data model, to describe both data and relevant metadata of aggregate objects. Moreover, the IAM manages a cache [14] of aggregates within the current unit of work (i.e., transaction). Specifically, the cache is intended to store a snapshot copy of aggregate objects, taken when they are read from the datastore, to support automatic

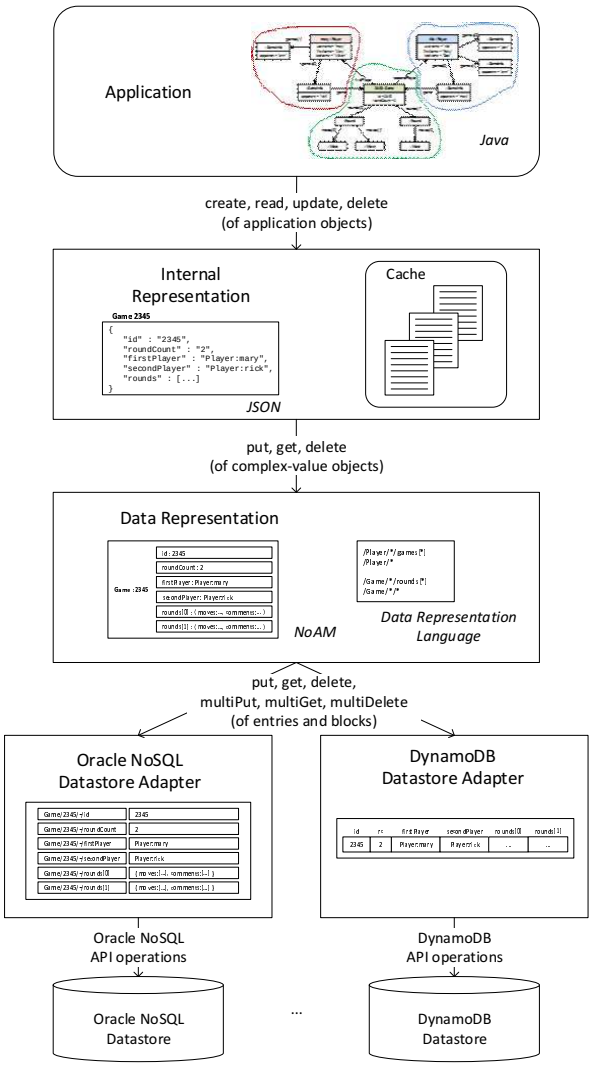


Figure 8: ONDM system architecture (abridged)

change tracking of application objects and the propagation of changes to the datastore when the unit of work commits.

The *Data Representation Manager* (*DRM*) handles the mapping between aggregates and the NoAM abstract data representation based on blocks and entries (such as the one shown in Fig. 3). The mapping is specified in terms of either basic data representation strategies or using the NoAM language for data representations [7].

Finally, data access is implemented by *Datastore Adapters* (*DA*s), each of which is specific for a certain target NoSQL system. Each datastore adapter is responsible for the conversion between the NoAM abstract data representation and the data representation in a specific datastore (such as the one shown in Fig. 4). It also transforms read-write operations on blocks and entries in operations for the specific system.

The description of a few usage scenarios will let us clarify the responsibilities of the various elements of the proposed system.

First, consider the creation of a new aggregate object. This operation is requested through the API. The internal

aggregate manager converts the in-memory aggregate object to the internal representation, and then passes the request to the data representation manager. The DRM transforms the aggregate into a set of NoAM entries, according to the data representation for the aggregate (specified by means of annotations through the API), as outlined in Section 2. Then, the DRM requests the storing of this set of entries to the proper datastore adapter (which is also specified by means of annotations). Finally, the selected DA converts the set of entries into specific data structures for the selected datastore, and requests their storage by means of specific system operations.

Then, consider the reading of an aggregate object, performed again through the API. The IAM passes the request to the data representation manager. The DRM converts the request into the reading of a set of NoAM entries, on the basis of the data representation for the aggregate. According to the selected representation, a single get operation or a multiGet operation is issued to the proper datastore adapter. After the corresponding operation has been executed by the target DA, the DRM converts the entry (or the set of entries, thereof) into an aggregate object (in the internal representation). Then, the internal aggregate manager converts this aggregate object in a group of in-memory application objects (possibly comprising an entity object and a number of value objects) and, before returning these objects to the application, puts a copy of them in the cache.

Finally, consider the update of an aggregate object. At commit time, the internal aggregate manager compares the updated aggregate object with the corresponding snapshot in the cache, to find possible changes that would require effective updates in the underlying datastore. (Recall that the cache contains a snapshot of each aggregate object, taken when they have been read from the datastore.) Such a comparison is handled by the data representation manager, and is executed with reference to the data representation for the aggregate, to track down differences at the level of individual entries. The difference for the aggregate object is determined in terms of modifications, additions, and deletions of entries. Then, the DRM uses such changes to request the correct sequence of operations to the datastore adapter, to update the entries associated with the aggregate object. The DA performs the requested operations, thus updating the representation of the aggregate object in the database.

The proposed architecture supports *data independence*, as follows. The application code is written with respect to the application programming interface, and is independent of both the target datastore and the data representation. The internal representation depends on how application objects are organized in entities and aggregates, but it is otherwise independent of the target datastore and the data representation. The NoAM data representation depends on the custom data representation, but it is independent of the organization of application objects in entities and aggregates, as well as of the target NoSQL datastore. Finally, each datastore adapter depends on a specific target NoSQL system, but it is independent of the data organization in the application and of the particular data representation. Table 1 summarizes the dependencies in ONDM. We can therefore say that ONDM provides data independence — the way in which application manages data is independent of the target datastore and of the selected data representation. This independence is fundamental in how our tool supports the flexible prototyping and experimentation as required in assessing NoSQL database design. This separation of concerns also supports extensibility of the tool itself.

	organization in entities and aggregates	data representation	target datastore
application program	yes	no	no
internal representation	yes	no	no
NoAM data representation	no	yes	no
datastore adapters	no	no	yes

Table 1: Dependencies in ONDM

4. APPLICATION PROGRAMMING INTERFACE

The application programming interface provides facilities for the following concerns: describing the organization of persistent application objects; specifying the target datastore and the data mapping towards it; and allowing programmers to perform CRUD operations on application objects. It is largely based on JPA [10], with a few variations.

The API adopts a conceptual entity model — based on entities, value objects (which are called *embeddable objects* in JPA), and relationships — whose constructs are associated with 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). Embeddable objects correspond to DDD's *value objects*. Fields representing relationships are annotated in a way depending on the multiplicity of the relationship, e.g., `@ManyToOne`. See Fig. 9.

In our approach, as we described in Section 2, application data should be organized in aggregates. Each *aggregate object* clusters a group of application objects (entities and value objects), having an entity as root [9]. In ONDM, we assume that each aggregate object includes only one entity object (its root), and that it possibly contains many value objects. Thus, in ONDM, annotation `@Entity` specifies also the root of an aggregate. Then, the complex value of each aggregate includes the root entity object and all the value objects that can be reached by navigating, directly or indirectly, non-relationship fields. Moreover, an aggregate includes also the references stored in fields representing relationships. Figure 1 shows a few sample application objects for our running example, and their composition in aggregates.

With respect to JPA, ONDM introduces two additional annotations, to describe entities and aggregates further: Annotation `@NoSql` states that an aggregate should be persisted in a NoSQL datastore (rather than in a relational database, as JPA usually does). `@NoSql` can include directives for a specific datastore. Annotation `@DataRepresentation` is used to specify the data representation for an aggregate; it can be either a basic representation strategy (such as

```
@Entity
@NoSql(datastore="OracleNoSQL")
@DataRepresentation(strategy="EAO")
class Player {
    @Id String userName;
    String firstName;
    String lastName;
    List<GameInfo> games;
}

@Embeddable
class GameInfo {
    String opponent;
    @ManyToOne Game game;
    ...
}

@Entity
@NoSql(datastore="OracleNoSQL")
@DataRepresentation(rules="*/rounds[*];*")
class Game {
    @Id long id;
    int roundCount;
    @ManyToOne Player firstPlayer;
    @ManyToOne Player secondPlayer;
    List<Round> rounds;
}

@Embeddable
class Round {
    List<Move> moves;
    List<String> comments;
    ...
}
```

Figure 9: Sample entity types (abridged)

EAO, Entity for Aggregate Object, described next) or a custom data representation, expressed as a set of rules in the NoAM language for data representations. (Data representations are discussed in Section 6.1.) Both `@NoSql` and `@DataRepresentation` should be used along with annotation `@Entity`.

Let us consider again the running example introduced in Section 2. We will use two entity types, `Player` and `Game`, which will also be the aggregate roots for our application. See Fig. 9. According to the mapping (see the `@NoSql` annotations), they will be persisted in Oracle NoSQL. Each aggregate `Player` object includes a `Player` entity object, together with a collection of `GameInfo` value objects. `Player` aggregates will be managed using the EAO (Entry per Aggregate Object) data representation strategy (see the `@DataRepresentation` annotation). EAO means that the whole complex value of each player will be represented using a single entry. Each aggregate `Game` object includes a `Game` entity object and multiple `Round` value objects (plus other `Move` objects, not specified in the figure). Aggregate `Game` will be managed using a custom data representation, specified by means of rules `*/rounds[*]` (an entry for each element of collection `rounds`) and `*` (an entry for the remaining data of the aggregate object). In practice, as shown in Fig. 3.

`GameInfo`, `Round`, and `Move` are embeddable classes, whose details can be safely ignored in this paper.

It is worth noting that, in ONDM, class and mapping metadata can be specified in two alternative but otherwise equivalent forms: by means of either annotations (as described so far) or XML configuration files. Using the latter form, metadata can be kept completely separate from application code.

```
public void addNewRound(long gameId, Round round) {
    /* em is the persistence manager */
    em.getTransaction().begin();
    Game game = em.find(Game.class, gameId);
    game.getRounds().add(round);
    game.incrementRoundCount();
    em.getTransaction().commit();
}
```

Figure 10: Usage of the persistence manager (abridged)

The API also defines a *persistence manager* interface, used by application developers to perform CRUD (Create, Read, Update, Delete) operations on aggregate objects. Figure 10 shows a representative usage of the persistence manager, to add a round to a game. The persistence manager is implemented by the underlying layers, described next.

The API allows developers to specify the target datastore and data representation, but it is otherwise independent of them.

5. INTERNAL AGGREGATE MANAGER

A main responsibility of the internal aggregate manager is the conversion between in-memory application objects and an internal representation for aggregates. The internal representation of aggregate objects is based on a complex-value data model [1], and adopts the popular data-interchange format JSON as syntax. For example, Fig. 11 shows the internal representation of the sample aggregate objects of Fig. 1 and 2, according to the entity types and the mapping specified in Fig. 9.

In the internal manager, each aggregate object has an individual and independent representation, including data about the root entity and all its associated value objects. Indeed, each value object is embedded in and represented together with the aggregate owning that element. Moreover, the internal representation of an aggregate object includes also a proper coding of references to other aggregate objects, in correspondence with relationship fields. A link to another object is thus represented either by embedding or by referencing the linked object, depending on whether the linked object is a value object or an entity. Thus, the internal representation is not a standard object serialization[1] in JSON, but rather a specific description of aggregates.

When the application needs to persist an aggregate object, this is first converted into the internal representation, then converted according to a selected data representation, and finally persisted in a datastore using a suitable adapter. On the other direction, when the application needs to retrieve an aggregate object, its data representation is first loaded from a datastore using an adapter, then it is converted into the internal representation, and finally it is converted into a group of in-memory application objects. The internal representation takes care of the boundaries of aggregate objects, but it is otherwise independent of any specific data representation and target datastore.

ONDM supports automatic change tracking, in the sense that application developers should not specify which indi-

[1]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.

```
Player:mary
{
   "username" : "mary",
   "firstName" : "Mary",
   "lastName" : "Wilson",
   "games" : [ { "game" : "Game:2345", "opponent" : "rick" },
               { "game" : "Game:2611", "opponent" : "ann" } ]
}

Player:rick
{
   "username" : "rick",
   "firstName" : "Ricky",
   "lastName" : "Doe",
   "games" : [ { "game" : "Game:2345", "opponent" : "mary" },
               { "game" : "Game:7425", "opponent" : "ann" },
               { "game" : "Game:1241", "opponent" : "john" } ]
}

Game:2345
{
   "id" : "2345",
   "roundCount" : "2",
   "firstPlayer" : "Player:mary",
   "secondPlayer" : "Player:rick",
   "rounds" : [ { "moves" : ..., "comments" : ... },
                { "moves" : ..., "comments" : ... } ]
}
```

Figure 11: Sample entities (internal representation in JSON)

vidual objects, and how, have been modified by each transaction. Therefore, developers can manage updates in a simplified way. To support this feature, the internal aggregate manager is also responsible of managing a *cache* of aggregate objects. As it is customary in ORM frameworks, the scope of the cache is the current unit of work (i.e., transaction). Specifically, we follow the *Implicit Copy-On-Read* approach [14]: the cache is intended to store a snapshot copy of aggregate objects, taken as soon as they are read from the datastore.

These snapshots are used, in particular, when the unit of work commits. In this case, the final states of in-memory objects are compared with their read-time snapshot states, to determine which objects have been indeed modified, to request just the needed update operations to the underlying datastore. The responsibility of this comparison is shared with the data representation manager, described in Section 7, to track down and perform updates at the granularity of individual entries.

6. NOAM

Before continuing the presentation of the architecture of ONDM, it is useful to describe NoAM in a little bit more detail. (We refer the reader to [7] for a detailed presentation of NoAM.) *NoAM* (*NoSQL Abstract Model*) is a system-independent approach for NoSQL database design. This method takes as input an application dataset of aggregate objects, maps it to an intermediate representation in an abstract data model, and then implements it according to the data structures of a target NoSQL system.

An *application dataset* includes a number of *aggregate classes*, each having a distinct name. The extent of an *aggregate class* is a set of *aggregate objects*. Each aggregate object has a *complex value* and an *identifier* (which is unique within the class the aggregate object belongs to). References are used to represent (unidirectional) relationships between aggregate objects. In our approach, we assume that aggregates have been already identified, and use them as input.

The NoAM *abstract data model* is intended to generalize the common aspects of the specific modeling features of the various NoSQL systems. It is used as an intermediate mapping model between application datasets of aggregate objects and NoSQL databases. This abstract model allows handling the initial activities of the design process in a system-independent way.

In NoAM, data are organized in *blocks* and *entries*, which represent units of data access and manipulation in NoSQL datastores at different granularity. Blocks are maximal units of consistency (such as documents in document stores and rows in extensible record stores). Entries are smaller units of data access (such as document fields and columns). In key-value stores, entries and blocks correspond to key-value pairs and groups of related key-value pairs, respectively. NoAM defines also *collections*, which represent, for example, document collections in document stores and tables in extensible record stores. The model includes also a set of data access operations, to specify the insertion, retrieval, update, and deletion of data elements in a system-independent way. Specifically, it defines operations to access a single entry, a whole block, or just a subset of the entries of a block.

In NoAM, each aggregate class is represented by means of a distinct collection, and each aggregate object by means of a block. The complex value of an aggregate object is represented by a set of entries in the corresponding block. For example, the dataset of aggregate objects of Fig. 2 can be represented by the NoAM database shown in Fig. 3.

The NoAM approach offers flexibility in the choice of the data representation. Indeed, given a dataset of aggregate objects, several representations are usually possible for them in the NoAM data model. Intuitively, each data representation is based on a partitioning of the complex values of aggregates in smaller data access units, each of which will be represented by a distinct entry. Intuitively, each entry represents a distinct portion of the complex value of an aggregate object, characterized by a location in its structure (called the entry key) and a value (the entry value). The choice of the entries for an aggregate object can be driven by the data access operations implied by the use cases of the application, on the basis of a number of guidelines [7].

It is worth noting that the guidelines for aggregate partitioning and the identification of entries are mostly heuristics, and so their application could lead to multiple alternative data representations. Given the relevance of this issue, we will discuss how NoAM allows the specification of data representations in Section 6.1.

Finally, the NoAM approach describes how to implement the identified data representation for the aggregates in a target NoSQL datastore. Given that the abstract data model generalizes the features of the various NoSQL systems, while keeping their major aspects, the implementation in specific datastores is rather straightforward. With reference to major NoSQL categories, we have that, in document stores, blocks and entries are mapped to documents and their fields, respectively. In extensible record stores, blocks and entries are mapped to records/rows and their columns, respectively. In key-value stores, entries are mapped to key-value pairs, while blocks are mapped to groups of related key-value pairs.

6.1 Data Representations

An application dataset of aggregate objects can be represented in the NoAM data model according to several alternative strategies. The various data representations differ in the choice of the entries used to partition and represent the complex value of each aggregate object.

A simple general data representation strategy, called *Entry per Aggregate Object* (*EAO*), represents each individual aggregate object using a single entry, whose value is the whole complex value of the aggregate object.

Another general representation strategy, called *Entry per Atomic Value* (*EAV*), represents each aggregate object by means of several entries, one for each atomic value in the complex value of the aggregate object.

A further general representation strategy, called *Entry per Top-level Field* (*ETF*), represents each aggregate object by means of multiple entries, using a distinct entry for each top-level field of the complex value of the aggregate object.

The above general data representation strategies can be suited in some cases, but they are too rigid and limiting in many other cases. In order to overcome these limitations, NoAM makes possible the specification of "customized" (i.e., user-defined) data representations, which are based on a flexible choice of entries for the aggregates. The main idea behind customization is that every data representation describes each aggregate object by means of a block composed of one or more entries. Hence, a customization can be specified by a partitioning of the complex values of aggregates. To this end, NoAM introduces a language, having an XPath-like syntax, to specify data representations. Let us start by showing how some general representation strategies can be specified using this language.

- Rule `/*/*` specifies strategy EAO (entry per aggregate object); the two stars refer to class names and aggregate object identifiers, respectively; we will have an entry for each distinct class and aggregate object.
- Rule `/*/*/*` specifies strategy ETF (entry per top-level field); the third star refers to top-level field names in the structure of complex values of aggregates; we will have an entry for each distinct class, aggregate object, and top-level field.

A single rule allows us to define a simple data representation, as in the above examples. However, a custom data representation can be specified by means of multiple rules. For example, the following sequence of rules leads to the data representation shown in Fig. 3:

- `/Player/*/games[*]` — uses an entry for each element of collection *games* of each **Player** aggregate object;
- `/Game/*/rounds[*]` — uses an entry for each element of collection *rounds* of each **Game** aggregate object;
- `/*/*/*` — specifies the use of strategy ETF for the remaining data.

7. DATA REPRESENTATION MANAGER

Let us go back to the presentation of the architecture of ONDM. The application programming interface deals with application objects (entities and aggregates), while the internal aggregate manager converts them into an internal representation.

Game:2345 : ⟨
id : *"2345"*,
roundCount : *3*,
firstPlayer : **Player:mary**,
secondPlayer : **Player:rick**,
rounds : {
⟨ *moves* : . . . , *comments* : . . . ⟩,
⟨ *moves* : . . . , *comments* : . . . ⟩,
⟨ *moves* : . . . , *comments* : . . . ⟩
}
⟩

Figure 12: An aggregate object after an update

The main responsibility of the data representation manager is handling the mapping between the internal representation of aggregate objects and the data representation in NoAM, based on collections, blocks, and entries, with reference to a general or a custom data representation, as described in Section 6.1. For example, with reference to the data representation specified in Fig. 9, from the internal representation shown in Fig. 11 to the data representation shown in Fig. 3 (when we need to persist an aggregate object), and viceversa (when we are reading an aggregate object).

Please note that, while the NoAM language is intended to specify the data representation for a whole application dataset, in ODNM data representations are specified individually for each class of aggregates, by means of the annotation `@DataRepresentation` provided by the API. Consider again the example shown in Fig. 9. There, the representation specified for aggregate `Player` is EAO; this is interpreted as a NoAM rule `/Player/*`. The representation specified for aggregate `Game` is `"*/rounds[*];*"`; this is interpreted as rules `/Game/*/rounds[*]` and `/Game/*`.

Recall from Section 5 that, to support automatic change tracking, the data representation manager is also responsible of performing the comparison of aggregate objects, as required when committing a unit of work. Specifically, it should discover modifications of application objects at a proper level of detail, to request the needed update operations to the underlying datastore. Here, the "proper level of detail" is, for each aggregate object, at the level of individual entries, which correspond to the finest data access units. Since the specific entries used to represent an aggregate object depend on the data representation, the identification of the differences is given to the data representation manager.

For example, consider an execution of the operation to add a round to a game, shown in Fig. 10. Assume that, as a consequence, the aggregate **Game:2345** of Fig. 2 has been modified as shown in Fig. 12, that is, a third round has been added and the round counter has been incremented. If the representation for games is the one shown in Fig. 3, then the data representation manager should discover, at commit time, that the update should be handled by adding an entry for the new round to the block for the game, and to modify the entry for the round counter, as shown in Fig. 13. In the figure, + and ∼ denote, respectively, an addition or a modification of an entry. However, if the data representation for games were EAO (Entry per Aggregate Object), then the update should be managed by a rewrite of the only entry in the block for the aggregate.

The data representation manager makes the above lay-

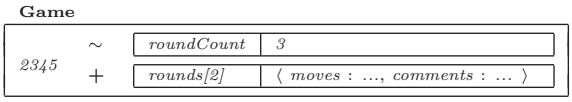


Figure 13: Updates required in the database

ers independent of the data representation for aggregates. It is also independent of specific target datastores, since it interacts with them through datastore adapters, described next.

8. DATASTORE ADAPTERS

A datastore adapter implements data access for a specific NoSQL datastore. ONDM has multiple adapters, one for each supported target system. The main responsibility of an adapter is the conversion between the NoAM abstract data representation (such as the one shown in Fig. 3) and the data representation in a specific datastore (such as the one shown in Fig. 4).

Each datastore adapter implements a common interface for the data access of blocks and entries, as required by the data representation manager. It then transforms read-write operations on blocks and entries in specific operations for the corresponding system.

Indeed, despite the differences in data models and APIs offered by the various NoSQL systems, there are also commonalities, which ONDM exploits in its datastore adapters. The implementation of datastore adapters take care of requesting the native data access operations offered by the corresponding system to provide efficient read-write operations of aggregate objects. According to our experience, the use of specific data access operations can improve performances by a factor up to 5 (over the use of basic read-write operations).

We will now describe, with respect to major NoSQL categories, three of the datastore adapters currently available in ONDM: for a key-value store (in some detail), and for extensible record stores and document stores (outlined). The conversions implemented by them are also described in [7].

Oracle NoSQL [17] is a key-value store. In Oracle NoSQL, a database is a schemaless collection of key-value pairs, with a key-value index. In Oracle NoSQL, *keys* are structured; they are composed of a *major key* and a *minor key*. The major key is a non-empty sequence of plain strings. The minor key is a sequence of plain strings; it can also be an empty sequence. Each element of a key is called a *component* of the key. Moreover, in Oracle NoSQL a *value* is an uninterpreted binary string.

The datastore adapter for Oracle NoSQL handles the mapping from a NoAM database (such as the one shown in Fig. 3) to an Oracle NoSQL database as follows. Each entry in the NoAM database is represented by an Oracle NoSQL key-value pair. Recall that each entry is a portion of the complex value of an aggregate object, which is represented, overall, by a block in some collection. The major key is composed of the name of the collection and the block key (i.e., the identifier of the aggregate object). The minor key is a proper coding of the entry key (that is, the location in the structure of the complex value of the aggregate object for the value represented by the entry). An example of key is */Player/mary/-/username*, where symbol / separates components, and symbol - separates the major key from the minor key. The value associated with this key is a representation of the entry value. It can be either a simple value or a serialization of a complex value, e.g., in JSON. Figure 4 shows the representation in Oracle NoSQL of the NoAM database of Fig. 3.

collection **Player**

id	*document*
mary	{ *username:"mary",* *firstName:"Mary",* *lastName:"Wilson",* *games[1]: { game:"Game:2345", opponent:"rick" },* *games[2]: { game:"Game:7425", opponent:"ann" }* }

Figure 14: Representation of the sample database of Fig. 3 in a document store (abridged)

A datastore adapter implements also the data access operations on blocks and entries. In this case, the retrieval of a block can be implemented either using a single Oracle NoSQL get operation (if the data representation for the specified aggregate corresponds to EAO), or using a single multiGet operation (if, otherwise, multiple entries are used within each block). The storage of a block can be implemented either using a single put operation (if the data representation corresponds to EAO), or using multiple put operations (otherwise). While Oracle NoSQL does not define a "write" counterpart of operation multiGet, it provides an operation execute for performing multiple put and delete operations in an atomic way — provided that the keys specified in these operations all share a same major key, which is indeed our case.

Document-oriented datastores, such as MongoDB [16], organize data in *collections* of documents. Each *document* is a structured document, that is, a complex value, a set of field-value pairs, which can comprise simple values, lists, and even nested documents. Documents are schemaless, as each document can have its own attributes, defined at runtime. A datastore adapter for a document store maps each collection of aggregates to a document collection, and each block to a single main document. The entries of the block are used as top-level fields in the document. See Fig. 14.

An extensible record store, such as Amazon DynamoDB [2] or Cassandra [3], organizes a database in tables. A *table* is a set of items. Each *item* contains one or more *attributes*, each with a *name* and a *value* (or a set of values). Each table designates an attribute as *primary key*. Items in a same table are not required to have the same set of attributes — apart from the primary key, which is the only mandatory attribute of a table. In ONDM, a datastore adapter for an extensible record store can handle the mapping from a NoAM database on the basis of a distinct table for each collection and a single item for each block, whose primary key is the block key for the aggregate. The entries for an aggregate object are mapped to the attributes of the item. See for example Fig. 15.

Datastore adapters are independent of the data representation of aggregates, and let the data representation manager be independent of the target datastore.

table **Player**

username	*firstName*	*lastName*	*games[1]*	*games[2]*	*games[3]*
"mary"	*"Mary"*	*"Wilson"*	{ *game*: ..., *opponent*: ... }	{ ... }	
"rick"	*"Ricky"*	*"Doe"*	{ *game*: ..., *opponent*: ... }	{ ... }	{ ... }

table **Game**

id	*roundCount*	*firstPlayer*	*secondPlayer*	*rounds[1]*	*rounds[2]*	*rounds[3]*
2345	*2*	**Player:mary**	**Player:rick**	{ *moves*: ..., *comments*: ... }	{ *moves*: ..., *comments*: ... }	

Figure 15: Representation of the sample database of Fig. 3 in an extensible record store (abridged)

9. A CASE STUDY

We will now discuss a case study in NoSQL database design, with reference to the running example introduced in Section 2. For the sake of simplicity, we just focus on the representation and management of aggregate objects for the games.

Data for each game include a few scalar fields (such as an identifier, the round counter, and references to the opponent players), and a collection of rounds. Each round consists of a group of partially structured data.

Two important operations over games are: (1) the retrieval of a game, which should read all the data concerning the game; and (2) the addition of a round to a game, which should also update the round counter.

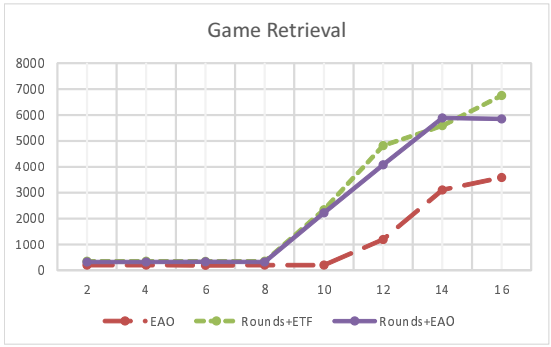
To manage games, we have identified three possible candidate data representations: (i) using a single entry for each game (that is, strategy EAO, as shown in Fig. 5, in the following called EAO); (ii) splitting the data for each game in a group of entries, and specifically using an entry for each round, and then an entry for each remaining top-level field (that is, on the basis of rules `/Game/*/rounds[*]` and `/Game/*/*`, as shown in Fig. 3, called Rounds+ETF); and (iii) a variant of representation (ii), but using an entry for all the remaining scalar fields (that is, rules `/Game/*/rounds[*]` and `/Game/*`, as shown in Fig. 6, called Rounds+EAO).

Assume also that we have chosen a key-value store as the target system to manage games.

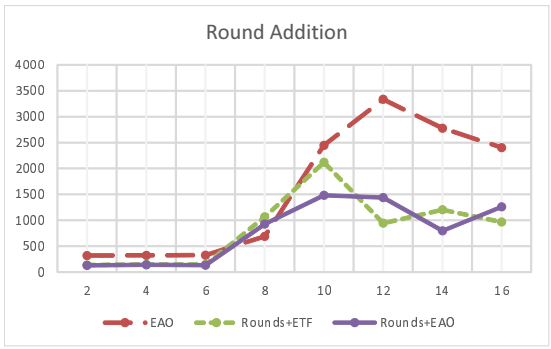
With respect to performance, we expect that the first operation (retrieval of a game) is favored by representation EAO, since it should read just a key-value pair, while the second operation (addition of a round to a game) is favored by representations Rounds+ETF and Rounds+EAO, which do not rewrite the whole game. Please note that this intuition does not give any quantitative estimate of the relative performance of the various alternatives. Thus, if we need to optimize performance, a few experiments are required. Specifically, we would like to evaluate the running time of a few application workloads with respect to the above data representations. However, we do not want to write completely different code for each alternative representation, but we would like to write the code just once, and then modify a few metadata in order to change the data representation in use.

ONDM fits this scenario, as it decouples the structure of the application data from the data organization in the datastore, mainly on the basis of the language for data representations, provided by the application programming interface (see Fig. 9).

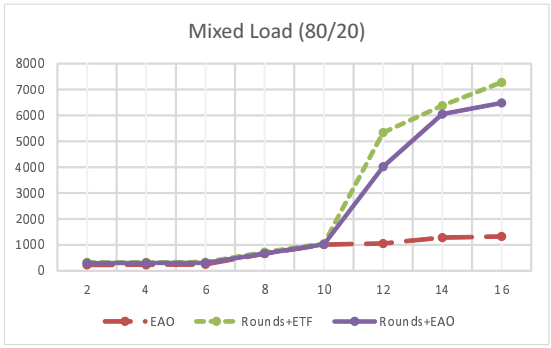
Therefore, we set up a few experiments, with a pseudo-random generator for application data with the following parameters: Each game has, on average, a dozen rounds, where each round is half a kilobyte circa. The remaining scalar data for a game amounts to a hundred bytes circa, for a total of 8KB per game, on average. At each run, we created $F \times 128,000$ games, for a total of F GB of application data. We then simulated the following workloads: (a) $F \times 64,000$ game retrievals (in random order); (b) $F \times 64,000$ round additions (to random games); and (c) a mixed workload, with $F \times 96,000$ game retrieval/round addition operations, with a read/write ratio of 80/20. We ran the experiments using different values of F (from 2 to 16 GB of data), to evaluate



(a) Game Retrieval (*msec per op.*)



(b) Round Addition (*msec per op.*)



(c) Mixed Load (*msec per op.*)

Figure 16: Experimental results

the three different data representations (EAO, Rounds+ETF, and Rounds+EAO) outlined above.

The target datastore was Oracle NoSQL, deployed over a single node, with 4 CPUs and 2GB of dedicated RAM.

The results are shown in Fig. 16. Timings are in milliseconds, and denote the average running time of a single operation.

As it is possible to see, the experiments confirmed the intuition that the retrieval of games (Fig. 16(a)) is always favored by representation EAO, for any database size. The advantage of EAO over the other representations increases when the database size is larger than 8 GB.

The experiments also showed that representation Rounds+EAO is generally better than representation Rounds+ETF, even if often just by a slight edge.

Moreover, the experiments concerning the addition of a round to an existing game (Fig. 16(b)) showed a general advantage of data representation Rounds+EAO over representation EAO, which is especially confirmed when the database size is larger than 8-10 GB.

Finally, the experiments over the mixed workload showed a general advantage of data representation EAO for our game application.

Thus, in this case, we can conclude that data representation EAO should be preferred, if 80/20 is indeed a representative read/write ratio.

10. RELATED WORK

The NoSQL arena is characterized by a high heterogeneity. Indeed, more than fifty NoSQL systems have been already implemented [20], each with different characteristics (e.g., different data models and different APIs to access the data, as well as different consistency and durability guarantees). Useful classifications of NoSQL datastores have been introduced in [8, 19]. In this paper we deal with aggregate-oriented databases, which include key-value stores, document stores, and extensible record stores. On the contrary, other datastore categories (such as graph databases) are beyond the scope of this paper.

In [5] it has been observed that, even in the NoSQL context, the availability of a high-level representation of the data remains a fundamental tool for designers and developers, since it makes understanding, managing, accessing, and integrating information sources much easier, independently of the technologies used. To this end, ONDM is based on NoAM [7], which introduces an intermediate, abstract data model that generalizes the common aspects of the specific modeling features of the various NoSQL datastores, that is independent of any specific system but suitable for each.

The development of methodologies and tools supporting NoSQL database design is demanding [5, 13, 15]. However, as far as we know, this topic has been not explored enough, and the related literature is very limited and focused on specific systems or system categories [11, 13]. A relevant exception is represented by the NoAM approach [7], which is a first effort in tackling the NoSQL database design problem from a general and system-independent perspective. It should be noticed that, as of today, NoSQL design methodologies are not yet mature enough to support all the design decisions needed in this context. As a consequence, practical experiments are then usually needed to identify a suitable design solution among various alternatives [19].

As we have said in the Introduction, our approach takes inspiration from Domain-Driven Design (DDD) [9], a widely followed object-oriented approach that includes a notion of aggregate. Moreover, [12] advocates the use of entities (which correspond to DDD's aggregates) as units of distribution and consistency. Also [19] refers to aggregate-oriented NoSQL databases, where the term "aggregate" comes from DDD.

The design and development of ONDM has been inspired by ORM (Object-Relational Mapping) frameworks [18]. In the context of NoSQL systems, ODM (Object-Datastore Mapping) is probably a better term than ORM. Some NoSQL systems provide developers with an ORM/ODM style data access, (e.g., Ohm[2]), and some ORM frameworks support, usually as a limited extension, a few NoSQL datastores (e.g., EclipseLink[3] and Hibernate Object/Grid Mapper[4]). There are also ODM frameworks that are specific to NoSQL datastores (e.g., Kundera[5]). To the best of our knowledge, the goal of the currently available ORM/ODM frameworks is mainly to easy the work of application developers, mostly by hiding the data structures and the application programming interface of specific systems. However, they do not provide tools to specify data representations in a flexible way. On the other hand, ONDM has the goal of both simplifying application development and enabling customized data representations.

In the scientific literature, [4] proposed SOS, a tool which provides a common programming interface towards different NoSQL systems, to access them in a unified way. For each target system, SOS defines a single data representation. In general, the definition of tools is complementary to the investigation of NoSQL-related issues, such as data access and database design.

We believe that ONDM is the first framework that aims at enabling the flexible prototyping and experimentation with different datastores, as required during NoSQL database design.

11. CONCLUSION

In this paper we have presented the design of ONDM (Object-NoSQL Datastore Mapper), a framework to access NoSQL datastores in a transparent and flexible way. It provides application developers with a uniform programming interface towards different systems. It also lets developers specify the desired mapping of the application data to a target NoSQL database in a declarative way. We believe that these features are needed to support NoSQL database design in the context of next-generation web applications.

At the time of writing, a first prototype of ONDM has been implemented. It provides the transparent access to a handful of NoSQL datastores, belonging to different categories, including Oracle NoSQL, Redis, Apache Cassandra, MongoDB, and Couchbase. For these systems, it implements a few basic data representation strategies, such as EAO, ETF, and EAV.

Currently, the architecture of this prototype is simpler than the one described in this paper. Specifically, it does not implement the NoAM language to specify custom data representations completely. An upgrade of the implementa-

[2] http://ohm.keyvalue.org
[3] http://www.eclipse.org/eclipselink/
[4] http://www.hibernate.org/subprojects/ogm.html
[5] https://code.google.com/p/kundera/

tion of the prototype is in progress, to include all the features described in this paper.

12. REFERENCES

[1] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[2] Amazon Web Services. DynamoDB. `http://aws.amazon.com/dynamodb`. Accessed 2013.

[3] Apache. Cassandra. `http://cassandra.apache.org`. Accessed 2013.

[4] P. Atzeni, F. Bugiotti, and L. Rossi. Uniform access to non-relational database systems: The SOS platform. In *Advanced Information Systems Engineering - 24th International Conference, CAiSE 2012*, pages 160–174, 2012.

[5] P. Atzeni, C. S. Jensen, G. Orsi, S. Ram, L. Tanca, and R. Torlone. The relational model is dead, SQL is dead, and I don't feel so good myself. *SIGMOD Record*, 42(2):64–68, 2013.

[6] C. Batini, S. Ceri, and S. B. Navathe. *Conceptual Database Design: An Entity-Relationship Approach*. Benjamin/Cummings, 1992.

[7] F. Bugiotti, L. Cabibbo, P. Atzeni, and R. Torlone. A logical approach to NoSQL databases. Submitted for publication, available at `http://cabibbo.dia.uniroma3.it/pub/noam.pdf`.

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

[9] E. Evans. *Domain-Driven Design*. Addison-Wesley, 2003.

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

[11] M. Hamrah. Data modeling at scale: MongoDB + mongoid, callbacks, and denormalizing data for efficiency. `http://blog.michaelhamrah.com/2011/08/data-modeling-at-scale-mongodb-mongoid-callbacks-and-denormalizing-data-for-efficiency/`, 2011.

[12] P. Helland. Life beyond distributed transactions: an apostate's opinion. In *CIDR 2007*, pages 132–141, 2007.

[13] I. Katsov. NoSQL data modeling techniques. `http://highlyscalable.wordpress.com/2012/03/01/nosql-data-modeling-techniques/`, 2012. Highly Scalable Blog.

[14] M. Keith and R. Stafford. Exposing the ORM cache. *ACM Queue*, 6(3):38–47, 2008.

[15] C. Mohan. History repeats itself: sensible and NonsenSQL aspects of the NoSQL hoopla. In *EDBT*, pages 11–16, 2013.

[16] MongoDB Inc. MongoDB. `http://www.mongodb.org`. Accessed 2013.

[17] Oracle. Oracle NoSQL Database. `http://www.oracle.com/technetwork/products/nosqldb`. Accessed 2013.

[18] C. Russell. Bridging the object-relational divide. *Queue*, 6(3):18–28, 2008.

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

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

[21] V. Vernon. *Implementing Domain-Driven Design*. Addison-Wesley, 2013.