A Logical Approach to NoSQL Databases

Francesca Bugiotti, Luca Cabibbo, Paolo Atzeni, Riccardo Torlone
Università Roma Tre, Italy
{bugiotti,cabibbo,atzeni,torlone}@dia.uniroma3.it

ABSTRACT

Although NoSQL databases are claimed to be “schemaless,” the design of data organization requires significant decisions. Specifically, persistent data of applications should be mapped to the modeling elements (collections, tables, documents, key-value pairs) available in the target system. These decisions are by no means trivial, because of their impact on major quality requirements, including scalability and performance, as well as consistency. Given the high heterogeneity in the NoSQL world, where more than fifty systems are available (each having different features), this design activity is usually based on best practices and guidelines which are strictly related to the selected system.

In this paper we present NoAM (NoSQL Abstract Model), a logical approach to the NoSQL database design problem, with initial activities that are independent of the specific target system. The approach aims at exploiting the commonalities of various NoSQL systems. It is based on an intermediate, abstract data model, which is used to represent application data as collections of aggregate objects (each having a complex value). Aggregates are units of distribution (to support scalability) and consistency (to the extent it is needed); aggregates can be partitioned in smaller data elements (for the sake of performance). We then show how the intermediate representation can be implemented in target NoSQL datastores, taking into account their specific features. The various implementations we propose, which refer to a number of representative NoSQL systems, provide an effective support for scalability, consistency, and performance.

1. INTRODUCTION

NoSQL datastores are a new generation of database systems that support the development of applications that require managing persistent data, but for which relational DBMSs are not well suited. According to [7, 21], a primary driver of interest in NoSQL datastores is their support for the scalability of next-generation web applications, that is, simple OLTP applications for which (i) a data access based on simple read-write operations is sufficient, (ii) data have a structure that does not fit well in the rigid structure of relational tables, and (iii) some quality requirements can be relaxed for performance and scalability (typically, strong consistency to eventual consistency).

Currently, more than fifty NoSQL systems exist [22], each with different characteristics (e.g., different data models and different APIs to access the data, as well as different consistency and durability guarantees). [7] suggests to classify them into a few categories — including key-value stores, document stores, and extensible record stores, plus others that are beyond the scope of this paper. As pointed out in [22], heterogeneity, even within each category, is highly problematic to application developers.

Although NoSQL datastores are claimed to be “schemaless,” the data of interest for applications do show some structure, which it may be useful to take advantage of. Indeed, persistent data of applications should be mapped to the modeling elements (collections, tables, documents, key-value pairs) available in the target system. Therefore, data organization in NoSQL datastores requires significant design decisions, as they affect major quality requirements, including scalability and performance, as well as consistency. This issue shows some similarities with problems studied in the database field in the past, for example in the logical design of relational databases [23] or to map XML documents to relational (or object-relational) storage [10].

Currently, database design in the NoSQL world is usually based on best practices and guidelines [13], which are specifically related to the selected system [20, 11, 18], with no systematic methodology. Indeed, several authors have observed that the development of high-level methodologies and tools supporting NoSQL database design are needed [4, 14, 13].

In this paper, we aim at filling this gap, by presenting NoAM (NoSQL Abstract Model), a logical approach to the NoSQL database design problem. The approach takes advantage of the commonalities of various NoSQL systems, thus including initial activities that are independent of the specific target system. So, it is based on an intermediate, abstract data model, to represent application datasets consisting of collections of aggregate objects, each having a complex value. The intermediate representation is then implemented in target NoSQL datastores, taking into account their specific features.

A major observation at the basis of our approach is that application data can be seen as arranged in aggregates [9, 12], that is, groups of related data, with a complex structure and value. Aggregates are natural units of data access and manipulation. They are also units of distribution (to support scalability) and consistency (to the extent it is needed) [9, 12]. An important issue with aggregates is that there is a trade-off in their design, which involves their granularity. On the one hand, each aggregate should be large enough, to include all the data involved by some integrity constraints or other business rules [24]. On the other hand, aggregates should be as small as possible, to reduce concurrency collisions and to support
performance and scalability requirements [24].

To deal with this issue in the context of NoSQL database design, we propose to consider each aggregate as a maximal unit of data access and consistency, and, for the sake of performance, to partition them into smaller data elements. Each of such elements is intended to be a minimal unit of data access and manipulation in the NoSQL database. This approach is compatible with the data model and access operations offered by most NoSQL datastores. Indeed, they usually provide operations that are atomic at a certain larger data granularity, as well as operations to manage data at a smaller grain. For example, the larger unit of data access in extensible record stores is the record, but it is also possible to access and manipulate smaller units corresponding to their individual columns. By partitioning aggregate objects into smaller data elements, we enable applications to perform data access and manipulation of aggregates in a finer way, thus improving performance and reducing concurrency collisions.

In accordance with the above observations, the NoSQL database design approach that we propose in this paper is intended to support the satisfaction of scalability, performance, and consistency requirements of the application. The NoAM approach is based on the following main phases:

- **aggregate design**, to identify the various classes of aggregate objects needed in an application; this activity is driven by use cases (functional requirements), as well as by scalability and consistency needs;
- **aggregate partitioning**, where aggregates are partitioned into smaller data elements; this activity is driven by use cases and performance requirements;
- **high-level NoSQL database design**, where aggregates are mapped to the NoAM intermediate data model, according to the identified partitions;
- **implementation**, to map the intermediate data representation to the specific modeling elements of a target datastore.

We point out that only the implementation phase depends on the target datastore. On the other hand, aggregate design and partitioning, as well as high-level NoSQL database design, are completely independent of the specific target system. In this sense, our approach is “logical” (that is, system independent).

A disclaimer is in order before moving forward. In this paper we cover the design of structures for NoSQL databases. It is important to observe that we do not mention important application qualities (such as availability, which can be pursued using data replication), which however are not neglected, as they are orthogonal to the issues we study. Moreover, we mainly refer to “simple” OLTP-like applications [7] and so we do not consider the case of applications that need to perform complex processing of large data sets, e.g., using MapReduce.

The paper is organized as follows: Section 2 provides an overview of the NoSQL database design problem and of our contribution. Section 3 presents the aggregate-oriented data model used at the application level and discusses aggregate design. Section 4 introduces the NoAM abstract data model for NoSQL databases, and presents a number of basic data representation strategies in our abstract data model. Section 5 introduces some guidelines to partition aggregates. Section 6 introduces data representations, and a language to specify them. Section 7 describes how a data representation can be implemented in some representative NoSQL systems. Section 8 discusses related work. Finally, Section 9 draws some conclusion.
than a complete rewrite of the whole game. This fact suggests that each aggregate object can be decomposed into smaller data access units, and it can be the case that operations refer to just one or a few of them. For example, in this phase we can decide to represent each round of a game as a separate data unit, as shown in Fig. 2.

The following step in the NoAM approach maps aggregates (and the smaller data units that partition them) into an abstract data model for NoSQL databases. As we said, applications of this kind are often implemented by using NoSQL datastores, since these systems are targeted to manage application datasets having the characteristics discussed above. That is, they provide operations that are atomic at a certain data granularity, as well as operations to manage data at a smaller grain. Indeed, in their data models, they usually define two distinct notions of data access “units” — a larger one and a smaller one. For example, in document stores, documents are the larger units, whereas their fields are the smaller units. In key-value stores, key-value pairs are the smaller units. In larger units are records/rows and their columns are the smaller units. In extensible record stores, larger units are records/rows and their columns are the smaller units. In key-value stores, key-value pairs are the smaller units, while data access is also possible on groups of records.

This observation suggests the definition of a data model that generalizes the common aspects of the specific modeling features of the various NoSQL systems. In NoAM, we call blocks and entries, respectively, the larger and the smaller data access units. An entry associates a key with a (possibly complex) value, while a block is a collection of entries.

We use this data model to handle a part of the design process in a system-independent way, by representing aggregates (and the smaller data units that partition them) in terms of blocks and entries. For example, Fig. 3 shows a possible representation of the aggregate objects of Fig. 1 in terms of the NoAM data model. There, outer boxes denote blocks correspond to aggregate objects, while inner boxes show entries.

In general, it is possible to represent a same application dataset according to different data representations. (See, for example, Fig. 7, 8, and 9.) To this end, NoAM introduces a language to specify data representations in a flexible way. Rules in this language let us describe the partitioning of aggregates that we have previously identified.

---

**Figure 3:** A sample database in the abstract model

---

**Figure 4:** Implementation in Oracle NoSQL for the sample database of Fig. 3

In the last phase of the NoAM approach, the selected data representation is implemented in the abstract model using the specific data structures of a target datastore. Given that the NoAM abstract model generalizes the features of the various NoSQL systems, while keeping their major aspects, it is rather straightforward to perform this activity. For example, if the target system is a key-value store, then each entry is mapped to a distinct key-value pair, while blocks correspond to groups of key-value pairs, sharing part of the key. For example, Fig. 4 shows how the abstract database of Fig. 3 can be mapped to Oracle NoSQL. An implementation can be considered “correct” if aggregate objects are indeed turned into units of distribution and consistency. The implementation shown in Fig. 3 is correct in this sense, by the use we make of Oracle NoSQL keys, which control distribution and atomicity of operations.

If the target system is an extensible-record store, such as DynamoDB, then each block is mapped to an item and each of its entries is mapped to an attribute of the item, as shown in Fig. 5. This implementation is also correct, since DynamoDB controls distribution and atomicity with reference to items.

To summarize, the contribution of this paper is in the definition of NoAM, a design approach that allows the specification of the organization of data in NoSQL databases, in a flexible way, according to application needs. This is first done in a system-independent way, and we also provide implementations in a number of representative datastores.

### 3. APPLICATION DATA MODEL

In this section we describe how data is organized and managed at the application level, according to a complex-value object model. Motivated by recognized principles and best practices in the design of scalable applications [9, 12], we consider application datasets organized in aggregates [21], with the following features:

- **An aggregate** [9] (called an `entity` in [12] and `entity group` in [5]) is a “chunk” of related data, with a complex structure and value, and a unique identifier. Each aggregate is designed to be a unit of data access and manipulation.

- Aggregates govern data distribution [12]. To support scalability, aggregates are distributed among the nodes of a computer cluster. Each aggregate object is located on a single node. Thus, aggregates are units of data distribution (i.e., sharding).
Aggregates are also units of atomic data manipulation. That is, atomicity is only provided to operations over single aggregate objects [12]. On the other hand, this property is not guaranteed to operations spanning multiple aggregate objects, which require multiple updates, each over a single aggregate. However, in this case eventual consistency [19] can be supported.

Before continuing with our presentation, let us now briefly discuss the characteristics of aggregates.

The fact that atomic operations can only involve a single aggregate object is a feature that is currently imposed by most NoSQL datastores. For example, Bigtable does not support transactions across rows [8], while MongoDB [15] provides only atomic operations over individual documents. This choice is motivated by scalability needs. Specifically, it has the goal of avoiding the coordination overhead required to guarantee atomicity of operations that would span multiple nodes in the cluster. This need is, in turn, implied by the fact that aggregates are units of data distribution. In general, most real applications require only operations that access a single aggregate object at a time [8].

Since atomicity is possible only within the boundary of an aggregate object, the enforcement of integrity constraints and other relevant business rules can only be guaranteed, in an atomic way, within the same boundary. Therefore, aggregates are also intended to define consistency boundaries [9].

To conclude, aggregates are natural units of data access and manipulation, and therefore they are the main focus of interest in our approach to NoSQL database design.

Let us now illustrate the terminology we use to describe data at the application level. An application dataset A includes a number of classes, each having a distinct name. The extent of a class is a set of aggregate objects. Each aggregate object has a complex value [1] and an identifier (which is unique within the class the aggregate object belongs to).

Values of aggregate objects are built using the following atomic values: (i) basic values, such as numbers, strings, and dates; and (ii) references, of the form C::id, where C is the class and id is the identifier of an aggregate object. References are used to represent (unidirectional) relationships between aggregate objects.

Moreover, values can be built using the following structured values:

- if $A_1, \ldots, A_n$ is a set of distinct names and $v_1, \ldots, v_n$ are values ($n \geq 0$), then $(A_1 : v_1, \ldots, A_n : v_n)$ is a record value; each $A_i$ is called a field of the record;
- if $v_1, \ldots, v_n$ are values ($n \geq 0$), then $(v_1, \ldots, v_n)$ is a collection value; each $v_i$ is called an element of the collection; each $i$ denotes the position (or index) of the corresponding element within the collection.

Specifically, we assume that the complex value of each aggregate object is a record. For the sake of simplicity, we assume that each class designates a top-level field to hold a basic value as the identifier for the aggregate objects in that class. Please note that, apart from classes and identifiers, our application datasets are almost seamless: in particular, the various aggregate objects in a class are not required to have a common structure.

### 3.1 Aggregate Design

The design of aggregates has the goal of identifying the classes of aggregates for an application, and various approaches are possible, to deal with both structured or only partially structured data.

For example, the approach proposed by Domain-Driven Design (DDD) [9], which is a widely followed object-oriented methodology, is driven by use cases (functional requirements), as well as by scalability and consistency needs. It proceeds as follows.

- The persistent data of an application are modeled, in a conceptual way [6], in terms of entities, value objects, and relationships. 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, entities and value objects are grouped in aggregates. Each aggregate has an entity as its root, and it 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.

It is worth noting that, because of their characteristics, the design of aggregates involves a trade-off concerning their granularity [24]. Aggregates should be large enough; each aggregate should include all the data involved by some integrity constraints or other forms of integrity constraint. On the other hand, aggregates should be as small as possible; small aggregates reduce concurrency collisions and support performance and scalability requirements.

In our running example, the online game application needs to manage various collections of objects, including players, games, and rounds. Figure 6 shows a few representative application objects. (There, boxes and arrows denote objects and links between

---

**Table Player**

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>mary</td>
<td>Mary</td>
<td>Wilson</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rick</td>
<td>Ricky</td>
<td>Doe</td>
<td>42</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Table Game**

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2/23</td>
<td>Player:mary</td>
<td>Player:rick</td>
<td>{ moves: ...</td>
<td>{ moves: ...</td>
<td>{ moves: ...</td>
</tr>
</tbody>
</table>

**Figure 6: Implementation in DynamoDB for the sample database of Fig. 3 (abridged)**

**Figure 6: Sample application objects**
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.) However, not all objects are considered as autonomous aggregates by themselves. Here, each player is an aggregate object, and also each game is an aggregate object, whose value includes (by nesting) the rounds played by the opponent players. We assume that the application should enforce the following consistency requirement: a round can be added to a game only if some condition that involves the other rounds of the game is satisfied. A game is an aggregate object, which can support the above business rule. On the other hand, a round cannot be an aggregate object by itself, since an individual round cannot check, alone, the above condition.

Thus the application dataset consists of aggregate classes Player and Game. Figure 1 shows, as complex values [1], two Player aggregate objects (having the username as identifier field) and a Game aggregate object (having id as the identifier field) played between them. The complex value of a Player object includes personal data on the player, as well as the collection of games she is currently playing; for each game, a summary comprising a reference to the Game object and a reference to the opponent Player object. The complex value of a Game object includes references to the Player objects involved in the game, as well as a collection of rounds. Please note that, by the rule of the game, each round comprises data that is only partially structured, and which can differ from round to round. Note also that the two Player objects have slightly different structure.

3.2 Handling Aggregate Objects

As it is customary, we assume that application developers manage aggregate objects by means of CRUD (Create, Read, Update, Delete) operations. In this scenario, a main goal of this paper is to describe how to represent aggregate objects in an underlying database, as well as how to implement CRUD data access operations over them. In doing so, we aim at supporting consistency, performance, and scalability, in a sense that we will discuss shortly.

Let us consider some data access operations for our sample application. When a player connects to the application, the aggregate object for the player should be retrieved, which comprises an overview of the games she is currently playing. Then, a player can select to continue a game, and the aggregate object for the selected game should be retrieved. Thus, as also pointed out in [8], most real applications require only operations that access a single aggregate object at a time.

When a player completes a round in a game she is playing, then the aggregate object for the game should be updated. To support performance, it is desirable that this update should be implemented in the database just as an addition of a round to a game, rather than a complete rewrite of the whole game. This fact suggests that aggregate objects should be represented in the underlying database in a way that enables the efficient manipulation of individual portions of aggregates, when needed. Therefore, we assume that data access operations handle single aggregate objects, or specific portions of them.

Furthermore, it is desirable to guarantee a suitable level of atomicity and consistency to update operations, since this can substantially simplify the job of application developers. Therefore, we consider ACID transactional operations over single aggregate objects, with the following observations. Individual aggregate objects are units of atomicity (an aggregate object is updated completely or it is not updated at all) and consistency (integrity constraints or other business rules can be defined over the boundary of single aggregate objects). We assume that isolation between transactions is based on versioning and optimistic concurrency control (as this is the kind of support usually provided by NoSQL systems). Specifically, each update of an aggregate object can be preceded by a reading of the aggregate object (to retrieve its current version), and then the update should fail if there is a version mismatch. For example, if a concurrent client has updated the same aggregate object before we completed our update (that is, a concurrency collision occurred).

We do not consider transactional operations spanning multiple aggregate objects. These have to be implemented as multiple operations, each over a single aggregate object. Atomicity, consistency, and isolation are not guaranteed, but eventual consistency can be obtained [19].

Summarizing, we assume only data access operations to single aggregate objects, or to specific portions of them, providing consistency, in the sense described above. We note also that scalability is supported by representing aggregate objects as units of data distribution [12]. Performance is also supported by designing aggregates as small as possible, since this also reduces the probability of concurrency collisions [24].

4. ABSTRACT DATA MODEL

In this section we define the NoAM abstract data model for NoSQL databases, which exploits the commonalities of their various data models, but also introduces abstractions to balance their differences and variations. As we will see in the remaining sections of this paper, this abstract data model will be used as an intermediate mapping model between application datasets of aggregate objects and NoSQL databases.

On the one hand, NoSQL datastores organize their data according to distinct data models — that differ in terms of their constructs and thus in how they structure data. Furthermore, these systems usually provide data access by means of simple read-write operations — that differ from system to system.

On the other hand, as we have observed in Section 2, NoSQL datastores share the common provision of having two distinct notions of data access “units” — a larger unit (which we call “block”), which is the maximal unit of consistency, and a smaller one (which we call “entry”), which is a unit of data access as well. Moreover, they often provide a notion corresponding to a “collection” of larger data units. With reference to major NoSQL categories, we have that, in document stores, blocks and entries correspond to documents and their fields, respectively. In extensible record stores, blocks and entries correspond to records/rows and their columns, respectively. In key-value stores, entries correspond to key-value pairs, while blocks correspond to groups of related key-value pairs.

We are now ready to present NoAM abstract data model for NoSQL datastores.

- A database is a set of collections. Each collection has a distinct name.

- A collection is a set of blocks. Each block in a collection is identified by a block key, which is unique within that collection.

- A block is a non-empty set of entries. Each entry is a pair \( \langle ek, ev \rangle \), where \( ek \) is the entry key (which is unique within its block) and \( ev \) is a complex value, called the entry value.

Intuitively, an application dataset can be represented by a database in this model as follows. We represent each class of aggregates by means of a distinct collection, and each aggregate object by means of a block. The complex value of each aggregate object is represented by a set of entries in the corresponding block. For example,
the application dataset of Fig. 1 can be represented by the database shown in Fig. 3. In the figure, inner boxes show entries, while outer boxes denote blocks. Collections are shown as groups of blocks.

The NoAM abstract data model includes also a set of abstract access operations, used to insert, retrieve, update, and delete data elements in an abstract database. Specifically, it defines operations to access a single entry, a whole block, or just a subset of the entries of a block. Indeed, the various NoSQL datastores offer operations corresponding to the above abstract operations. All such operations are usually provided as atomic operations (with a precaution for key-value stores, discussed in Section 7).

4.1 Examples of Usage

Given an application dataset of aggregate objects, several alternative approaches are usually possible to define a database in NoAM abstract data model to represent it. This section is devoted to the discussion of basic data representation strategies, which we illustrate with respect to the example described in Fig. 1. Additional and more flexible data representations will be introduced in Section 6.

As we have said before, in NoAM each class of aggregates is represented by means of a distinct collection, and each aggregate object is represented by means of a block. We use the class name to name the collection, and the identifier of the aggregate object as block key. 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. The entry key is empty. The entry value is the whole complex value of the aggregate object. The representation of the aggregate objects of Fig. 1 according to this representation strategy is shown in Fig. 7.

- A second general representation strategy, called Entry per Atomic Value (EAV), represents each individual aggregate object by means of multiple entries, whose values are atomic values. Specifically, it employs an entry for each atomic value contained in the complex value $v$ of the aggregate object. The entry key associated with each atomic value $v'$ is a coding of the path to access that component value $v'$ in $v$. (In Section 5 we will introduce “access paths” to formalize this idea.) Figure 8 shows the representation of the aggregate objects of Fig. 1 according to this representation strategy. Here we have several entries for each block. For example, there are seven of them for the Player object having username mary, while strategy EAO used only one.

Strategies EAO and EAV are the two most “extreme” data representations we can think of. On the one hand, to represent an aggregate object, EAO adopts a block comprising a single entry, where the key is a very simple key (the empty key is the simplest one) and the value is a very complex value (the most complex). On the other hand, to represent an aggregate object, EAV adopts a block comprising multiple entries, whose keys are the most complex and the values are the simplest.

With this in mind, it is possible to develop other “intermediate” data representations, each of which represents an aggregate object using a different group of entries, on the basis of some “decomposition” of its complex value.

- An intermediate 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 object. Specifically, for each top-level field $f$ of an aggregate object $o$, it employs an entry having as value the value of field $f$ in the complex value of $o$ (please note that such values can be complex values themselves), and as key the field name $f$. Figure 9 shows the representation of the aggregate objects of Fig. 1 according to this representation strategy. Note that, in this case, to
represent the Player object having username mary, we need one block having four distinct entries, corresponding to fields username, firstName, lastName, and games.

As a comparison, we can say that strategy ETF is not as “extreme” as strategies EAO and EAV. The keys of the entries in ETF are simple, but they are not the simplest possible keys. Values in ETF are complex values, but they are slightly less complex in structure than in EAO. We can therefore say that it is an “intermediate” representation strategy. Moreover, it is possible to note that EAO does not depend on the structure of aggregate objects, ETF depends on the top-level structure of objects (which can be “almost fixed” within each class), while EAV strongly depends on the specific complex structure of each object.

Although the above general data representation strategies can be suited in some cases, they can be too rigid and limiting in many cases. For example, none of the above strategies leads to the data representation shown in Fig. 3. Such data representation is a variant of strategy ETF (an entry for each top-level field) such that, if a field is a collection of records, then each record/element of such a collection is represented by a distinct entry. More precisely, this data representation adopts an entry for each element belonging to a top-level collection field, and then, for the remainder of the data, an entry for each top-level field (as in ETF). The entry value is the value of the selected element (being it a collection element or a top-level field element). The entry key locates the element in the structure of the aggregate object. The application of this data representation to the aggregate objects of Fig. 1 is indeed that in Fig. 3. Note that, in this case, to represent the Player object having username mary, we need five distinct entries, three of which correspond to fields username, firstName, and lastName, and the other two correspond to the individual games she is playing. Thus, this representation can be considered intermediate as well.

5. AGGREGATE PARTITIONING

In practice, we need more flexibility in the representation of aggregates using entries than the general strategies in Section 4.1 provide. In the NoAM abstract data model, each aggregate object is represented by a block, composed of one or multiple entries. An important intuition in our approach is that we can define a data representation in terms of a decomposition (specifically, a partitioning) of the complex value of each aggregate object into a group of entries. As we will discuss shortly, to identify the entries for each aggregate, it is useful to reason at the application level, with reference to operations implied by use cases, rather than just on the structure of objects.

We first need to introduce a preliminary notion of “access path,” to specify a “location” in the structure of a complex value, and to access the corresponding component value. Intuitively, if \( v \) is a complex value and \( \alpha \) is a value (possibly complex as well) occurring in \( v \), then the access path \( \alpha p \) for \( \alpha ' \) in \( v \) represents the sequence of “steps” that should be taken to reach the component value \( \alpha ' \) in \( v \). Conversely, if \( v \) is a complex value and \( \alpha p \) is an access path, then \( \alpha p(v) \) denotes the component value identified by \( \alpha p \) in \( v \).

An access path \( \alpha p \) is a (possibly empty) sequence of access steps, \( \alpha p = p_1 p_2 \ldots p_n \), where each step \( p_i \) identifies a component value in a structured value. Given a complex value \( v \), the access paths for \( v \) are defined, inductively, as follows:

- the empty access path \( \epsilon \) is an access path for \( v \) that accesses the whole value \( v \), that is, \( \epsilon(v) = v \);
- if \( \alpha p \) is an access path for \( v \) such that \( \alpha p(v) \) is the record value \( \langle A_1: v_1, \ldots, A_i: v_i, \ldots, A_n: v_n \rangle \), then \( \alpha p.A_i \) is an access path for \( v \) that accesses value \( v_i \), that is, \( \alpha p.A_i(v) = v_i \); step \( A_i \) is a called a field step;
- if \( \alpha p \) is an access path for \( v \) such that \( P(v) \) is the complex value \( \{v_1, \ldots, v_i, \ldots, v_n\} \), then \( \alpha p[i] \) is an access path for \( v \) that accesses value \( v_i \), that is, \( \alpha p[i](v) = v_i \); step \( i \) is called a position step.

Let us indicate with \( v_{mary} \), the complex value of the Player object having username mary shown in Fig. 1. Examples of access paths for this complex value \( v_{mary} \) are username and games[1].opponent. If we apply the latter access path to \( v_{mary} \), we can access value Player:rick.

In NoAM, we represent the complex value \( v \) of an aggregate object using a set of entries, whose keys are access paths for \( v \). Intuitively, each entry is intended to represent a distinct portion of the complex value \( v \), characterized by a location in its structure (the access path, used as entry key) and a value (the entry value).

With reference to our running example, a possible entry for the complex value \( v_{mary} \) of the Player object having username mary shown in Fig. 1 is the pair (games[1].opponent, Player:rick).

We have already applied the above intuition in Section 4.1. For example, data representation strategy ETF uses field names as entry keys, and they are indeed a case of access paths.

An important observation is needed. Consider a complex value \( v \) and an entry \( e = (\alpha p(e), v) \) for \( v \). It is possible that the entry value \( v \) is equal to \( \alpha p(e) \), that is, to the value that can be accessed in \( v \) by means of the access path \( \alpha p \), as shown in the example above. However, this is not required by our intended usage of entries. Indeed, as we will see, it is also possible that an entry value for \( v \) is just a portion of the complex value that can be reached by means of the corresponding access path.

We will now introduce the notion of partition of a complex value \( v \) as a set \( E \) of entries that fully cover the complex value \( v \), without redundancy. To formalize this notion, we need to consider complex values as trees.

In general, each complex value \( v \) can be viewed as a tree \( T(v) \) [1] (with the proviso of considering references just as atomic values). In such a tree, data are located in the leaves only, while the internal
nodes are intended to describe the structure of the complex value. According to this viewpoint, an access path in a complex value $v$ can be considered as the sequence of edges that should be traversed from the root to reach a certain node in $T(v)$.

Then, an entry $e$ for a complex value $v$ is intended to represent a sub-graph of $T(v)$ that is a tree. (Please note that the tree $T(v)$ for an entry $e$ for $v$ is not required to be a sub-tree of $T(v)$.)

Finally, a partition $E$ of a complex value $v$ is a set of entries such that each leaf in $T(v)$ (which represents an atomic value in $v$) appears in one and only one entry $e$ in $E$.

The choice of the entries for an aggregate object can be driven by the data access operations implied by the application use cases, using the following guidelines (which are a variant of guidelines proposed in [6]):

- If an aggregate is small in size, or all or most of its data are accessed or modified together, then it should be represented by a single entry.
- Conversely, an aggregate should be partitioned in multiple entries if it is large in size and there are operations that frequently access or modify only specific portions of the aggregate.
- Two or more data elements should belong to the same entry if they are frequently accessed or modified together.
- Two or more data elements should belong to distinct entries if they are usually accessed or modified separately.

In our sample application, data for each individual round is always read or written together. Moreover, data for the rounds of a game can be read together, but each round is always written separately. Therefore, each round is a candidate to be represented by an autonomous entry.

The application of the above guidelines can suggest a partitioning of aggregates, which we would like to use to guide the representation in the target datastore. To this end, it is useful to be able to specify candidate data representations.

6. DATA REPRESENTATIONS

In Section 4.1 we have exemplified the use of the NoAM abstract data model to represent aggregates. We have shown some general representation strategies (that is, EAO, ETF, and EAV), as well as an ad-hoc data representation (Fig. 3). Then, in Section 5 we have suggested to partition aggregate data in terms of smaller data elements, which can be identified by analyzing application use cases.

We now introduce a language to specify data representations in a flexible way, to describe a choice of the partitioning of aggregates.

The main idea is that every data representation models each aggregate object using a block composed of one or more entries of the NoAM abstract data model. Hence, it can be specified by describing a partitioning of the complex values of aggregates. Since aggregates and their complex values can be considered as trees, and also an application dataset of aggregates can be viewed as a tree, we use a language to select subsets of the nodes of a tree to specify how to partition aggregates.

For example, strategy EAV (entry per atomic value object) partitions a dataset tree with respect to its leaves, and this corresponds to the selection of all the leaf nodes of the tree. On the other hand, strategy EAO (entry per aggregate object) partitions a dataset tree with respect to individual aggregate objects, selecting the nodes for the roots of the complex values of the aggregate objects.

An application dataset $A$ of aggregate objects can be considered as a dataset tree $T(A)$, as follows. The root node $n_A$ represents the whole dataset $A$. Then, for each class $C$ of aggregates in the dataset we have a node $n_C$ which is a direct child of $n_A$. Finally, for each aggregate object $o$ belonging to a class $C$ and having complex value $v_o$, the tree contains a sub-tree $T(v_o)$ for $o$, representing the complex value of $o$; the root node $n_o$ of $T(v_o)$ is a direct child of the node $n_C$ for the class $C$ of $o$. For example, see Fig. 12.

6.1 The Language

We now introduce a language to specify data representations, based on a selection of the nodes of the dataset tree; these are the nodes that should contribute to entries in the abstract data model. To this end, we use an XPath-like syntax, which we will illustrate by means of examples. Let us start by showing how to specify the basic representation strategies of Section 4.1.

- 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.
- Rule $// * */ * !$ specifies strategy EAV (entry per atomic value); term $// * !$ denotes a full traversal of the structure of complex values of aggregates, but selecting only their leaves, which contain atomic values; we will have an entry for each distinct class, aggregate object, and atomic value.

A single rule lets us define a simple and general data representation, as in the above examples. Furthermore, more complex representations can be specified by means of multiple rules. The following rules lead to the data representation shown in Fig. 10:

- Rule $//Player/*$ specifies the adoption of strategy EAO for the aggregate objects of class Player; we will have an entry for each distinct aggregate object in class Player.
- Rule $//Game/*$ denotes the adoption of strategy ETF for the aggregate objects of class Game; this will lead to an entry for each top-level field of each object in class Game.

In general, a data representation can be specified as a sequence of rules. As we will describe soon, the ordering on the rules is important, as it defines precedence. Moreover, we require that the last element of a sequence corresponds to a general data representation strategy, that should be applied to cases that are not covered by other rules.

Consider again the data representation shown in Fig. 3. It can be specified by means of the following sequence of rules:

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

As a further example, assume that we want an entry for each element of the collection games of a Player and then an entry for the rest of the Player data. For the sake of simplicity, we consider here only class Player. This can be specified as follows:
Player

mary e (username:"mary", 
  firstName:"Mary", 
  lastName:"Wilson", 
  games: { 
   (game : Game:2345, opponent : Player:rick ), 
   (game : Game:2611, opponent : Player:ann) } )

rick e (username:"rick", 
  firstName:"Ricky", 
  lastName:"Doe", 
  score:42, 
  games: { 
   (game : Game:2345, opponent : Player:mary ), 
   (game : Game:7425, opponent : Player:ann), 
   (game : Game:1241, opponent : Player:johnny) } )

Game

id 2345
firstPlayer Player:mary
secondPlayer Player:rick
rounds { (moves: ..., actions: ..., spell: ... ) }

Figure 10: A sample data representation

Player

mary e (username:"mary", 
  firstName:"Mary", 
  lastName:"Wilson")

Games1] (game : Game:2345, opponent : Player:rick )
Games2] (game : Game:2611, opponent : Player:ann)

rick e (username:"rick", 
  firstName:"Ricky", 
  lastName:"Doe", 
  score:42, 
  games1] (game : Game:2345, opponent : Player:mary )
Games2] (game : Game:7425, opponent : Player:ann)
Games2] (game : Game:1241, opponent : Player:johnny)

Figure 11: Another sample data representation (abridged)

- /Player/*
- /Player/*

In applying these rules to the complex value of a Player object, we first apply the first rule, /player/*/*/* (EF), which has highest precedence; it captures all the data concerning the games played by the player. When we apply the second rule, /player/*/*/*/*, it should not be applied to the whole complex value of the Player, but just to the data that has not been captured by the application of previous rules. In this case, this leads to the identification of an entry for each player containing all its personal data (e.g., its username and name) but not data concerning her games. This representation is shown in Fig. 11.

6.2 Semantics

Let S be a sequence of rules, intended to specify a data representation. Moreover, let A be an application dataset. We will now define the semantics of S(A), as the database D that represents A according to S. Let T(A) be the dataset tree for A.

We first define the set of nodes in T(A) selected by a rule and a set of rules. If r is a rule, we denote by r(A) the set of nodes in T(A) that has been selected by evaluating rule r over A. For the evaluation of r(A) we refer to an XPath-like semantics, omitting the details for the sake of space. We simply assume that such a set $r(A)$ never contains any node in the first two levels of $T(A)$ (that is, neither the root nor nodes representing classes). We also assume that $r(A)$ never contains both a node $n$ in $T(A)$ and a proper descendant (or a proper ancestor) of the same node $n$ in $T(A)$.

The nodes selected by some representative rules are shown in Fig. 12. For the sequence S of rules, we define $S(A) = \bigcup_{r \in S} r(A)$ as the set of all nodes in $T(A)$ that have been selected by the individual rules in S.

It turns out that not every sequence $S$ of rules defines an effective data representation. Let $n$ be a node in $T(A)$. We denote by $L(n)$ the set of leaves of $T(A)$ that are descendants of $n$; we say that $n$ sees the leaves $L(n)$ of $T(A)$. For a set $N$ of nodes, we define

$$L(N) = \bigcup_{n \in N} L(n),$$

as the leaves seen by $N$. We say that a data representation $S$ is complete with respect to the application dataset $A$ if it is the case that $L(S(A))$ is equal to the set of all leaves of $T(A)$, that is, if $S(A)$ sees all the leaves of $T(A)$.

Moreover, let $n$ be a node and $N$ be a set of nodes. We say that $N$ sees at least as many as $n$ if $L(n) \subseteq L(N)$, that is, if the leaves seen from $N$ contain the leaves seen from $n$.

Let S be a sequence of rules, and $r, r'$ be rules in $S$. If $r$ precedes $r'$ in $S$, and thus $r$ has precedence over $r'$, we write $r < r'$. The precedence over rules induces a precedence over nodes, as follows. For each node $n$ in $S(A)$, we denote by $R(n)$ the set of rules in $S$ that select $n$ in $T(A)$. Moreover, we denote by $\pi(n)$ the rule with highest precedence in $R(n)$. Let $n, n'$ be nodes in $S(A)$; we say that $n$ has precedence over $n'$, written $n < n'$, if $\pi(n) < \pi(n')$. Please note that it is possible that $n$ and $n'$ have the same precedence, $n \approx n'$.

Now, $S(A)$ is a subset of the nodes of $T(A)$, with a precedence order defined on the nodes. We will now define the NoAM database $D$ that represents $A$ according to $S$. For the sake of our development, it is useful to consider the database $D$ as a “flattened” set of entries, where each entry has a qualified key and a value. The qualified key is composed of the collection name, the block key, and the entry key.

Each node $n$ in $S(A)$ can contribute to an entry $\langle k_n, v_n \rangle$ in $D$. If this is the case, then $k_n$ is the qualified entry key that represents the path from the root of $T(A)$ to node $n$. The entry value $v_n$ is, at most, the complex value that represents the sub-tree $T_n$ of $T(A)$ rooted in $n$. However, a node $n$ could also contribute to $v_n$ with a smaller sub-tree. It is also possible that $n$ does not contribute at all to an entry for $D$. Indeed, $v_n$ corresponds to the sub-tree $T_n$ rooted in $n$, pruned by all leaves and sub-trees that are seen from nodes in
that need to update just a subset of the entries of the block for an aggregate object. Since aggregates should be units of atomicity and consistency, if these operations are requested concurrently on the same aggregate object, then the application would require that the datastore identifies a concurrency collision, commits only one of the operations, and aborts the other. However, if the operations update two disjoint subsets of entries, then a key-value store is unable to identify the collision, since it has no notion of block. We support this requirement, thus providing atomicity and consistency over aggregates, by including in each update operation the access to a distinguished entry of the block for an aggregate object; in particular, this could be the entry that includes the identifier of the aggregate, or a specific “version” field.

7.2 Extensible Record Store: DynamoDB

Amazon DynamoDB [2] is a NoSQL database service provided on the cloud by Amazon Web Services (AWS). DynamoDB is an extensible record store. A database is organized 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. Thus, DynamoDB databases are mostly schemaless.

DynamoDB guarantees atomicity at the item level. Distribution is also operated at the item level, and controlled by a specific portion of primary keys.

With respect to the NoAM abstract data model, a natural representation for a database in DynamoDB is based on a distinct table for each collection, and a single item for each block. The item contains a number of attributes, which can be defined from the entries of the block for the item.

We now show how to implement a data representation $S$ in DynamoDB. Consider a block $b$ in a collection $C$ having block key $id$. According to $S$, one or multiple entries are used within each block. We will use all the entries of a block $b$ to create a new item in a table for $b$. Specifically, we proceed as follows: (i) the collection name $C$ is used as a DynamoDB table name; (ii) the block key $id$ is used as a DynamoDB primary key in that table; (iii) the set of entries (key-value pairs) of a block $b$'s is used as the set of attribute name-value pairs in the item for $b$ (a serialization of the values is used, if needed). For example, Fig. 5 shows the implementation of the data representation of Fig. 3.

In Oracle NoSQL the major key controls distribution (sharding is based on it) and consistency (an operation involving multiple key-value pairs can be executed atomically only if the various pairs are over a same major key).

With Oracle NoSQL, and more in general with key-value stores, a precaution is needed with update operations that access only a subset of the entries of a block. Consider two separate operations that need to update just a subset of the entries of the block for an aggregate object. Since aggregates should be units of atomicity and consistency, if these operations are requested concurrently on the same aggregate object, then the application would require that the datastore identifies a concurrency collision, commits only one of the operations, and aborts the other. However, if the operations update two disjoint subsets of entries, then a key-value store is unable to identify the collision, since it has no notion of block. We support this requirement, thus providing atomicity and consistency over aggregates, by including in each update operation the access to a distinguished entry of the block for an aggregate object; in particular, this could be the entry that includes the identifier of the aggregate, or a specific “version” field.

7.1 Key-Value Store: Oracle NoSQL

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. On the other hand, an Oracle NoSQL value is an uninterpreted binary string.

A data representation $S$ can be implemented in Oracle NoSQL as follows. We use an Oracle NoSQL key-value pair for each entry in $D$. The major key is composed of the collection name $C$ and the block key $id$, while the minor key is a proper coding of the entry key (that is, the access path $ap$ is coded using a distinct key component for each step in $ap$). 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 value $v$. It can be a simple value or a serialization of a complex value, e.g., in JSON.

The retrieval of a block can be implemented either using a single Oracle NoSQL get operation (if the data representation for the specified class corresponds to EAO), or using a single multiGet operation (if, otherwise, multiple entries have been used within each block). The storage of a block can be implemented either using a single put operation (if the data representation for the specified class 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.

For example, Fig. 13, 14, and 15 show the implementation of data representation strategies EAO, ETF, and EAV, respectively, in Oracle NoSQL. Moreover, Fig. 4 shows the implementation of the data representation of Fig. 3.

The effectiveness of this implementation is based on the fact that
7.3 Document Store: MongoDB

MongoDB [15] is an open-source, document-oriented data store. In MongoDB, a database is made of 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. A main document is a top-level document with a unique identifier, represented by a special field _id. Documents are schemaless, as each document can have its own attributes, defined at runtime. Specifically, MongoDB documents are based on BSON (Binary JSON), a variant of the popular JSON format.

According to our approach, a natural implementation for an abstract database in MongoDB is based on a distinct MongoDB collection for each collection of blocks, and a single main document for each block. The document for a block b has the JSON/BSON serialization of the complex value of b as value, plus a special field to store the block key id of b, as required by MongoDB, _id: id.

The NoAM data representation that more closely corresponds to this natural implementation is EAV. Consider a block b in a collection C having key id. According to such data representation strategy, multiple entries are used: an entry for each atomic value. We will use all the entries in block b to create a new document. Specifically, we proceed by building a document d for b as follows: (i) the collection name C is used as the MongoDB collection name; (ii) the block key id is used for the special top-level id field _id: id of d; (iii) then, each entry (key-value pair) in the block b is used to fill a (possibly nested) field of document d. See Fig. 16.

The retrieval of a block, given its collection C and key id, can be implemented by performing a find operation, to retrieve the main document that represents all the block (with its entries). The storage of a block can be implemented using an insert operation, which saves the whole block (with its entries), in an atomic way. It is worth noting that, using other MongoDB operations, it is also possible to access and update just a subset of the entries of a block, in an atomic way.

It is also possible to provide a different implementation for MongoDB, which represents a block b using again a main document for b, but that uses a distinct top-level field-value pair for each entry in our representation. In particular, for each entry (ek, ev), the document for b contains a top-level field whose name is a coding for the entry key (access path) ek, and whose value is either an atomic value or an embedded document that serializes the entry value ev. For example, according to this implementation, the data representation of Fig. 3 leads to the result shown in Fig. 17.

8. RELATED WORK

To the best of our knowledge, NoAM is the first proposal of a general and system-independent approach to the design of NoSQL databases. Indeed, although several authors have already observed that the development of methodologies and tools supporting NoSQL database design is demanding [4, 14, 13], as far as we know this topic has been not explored yet and the related literature is a very limited. The only examples are some on-line papers, usually published in blogs of practitioners, that discuss best practices and guidelines for modeling NoSQL databases, most of which are suited only for specific systems. For instance, [13] lists some techniques for implementing and managing data stored in different types of NoSQL systems, while [16] discusses design issues for the specific case of key-value datastores. On the system-oriented side, [20, 11, 18] illustrate design principles for the specific cases of HBase, MongoDB and Cassandra, respectively. None of them tackles the problem from a general perspective, as we try to do in this paper.

The issue of representing application data in a NoSQL datastore shows some similarities with problems studied by the database community in the past, such as (i) the early works on vertical partitioning and clustering [23], with the idea to put together the attributes that are accessed together and to separate those that are visited independently, and (ii) the more recent approaches to relational (or object-relational) storage of XML documents [10], where various alternatives obviously exist, with tables that can be very small ("binary") and handle individual edges, or very wide ("universal") and handle entire paths, from root to leaves—and many alternatives in between.

Our approach takes inspiration from Domain Driven Design [9], a widely followed object-oriented approach that includes a notion of aggregate. Moreover, [12] advocates the use of aggregates (there called entities) as units of distribution and consistency. However, we also propose to partition aggregates, to identify, for efficiency purposes, smaller units of data access and manipulation. This is coherent with the way in which most NoSQL systems manage data. Entity groups proposed in [5] are analogous to our aggregates, since an entity group is a set of entities that can be manipulated in an atomic way. Then, [5] describes a specific mapping of entity
groups to Bigtable [8]. Our approach is based on more abstract data models and is system independent, as it is targeted to a wide class of NoSQL systems.

In [4] it has been observed that the availability of a high-level representation of the data remains a fundamental tool for developers and users, since it makes understanding, managing, accessing, and integrating information sources much easier, independently of the technologies used. This is in line with the NoAM approach, which is based on an intermediate, abstract data model that, as it happens in the relational setting, makes it possible to devise an initial phase of the design process that is independent of any specific system but suitable for each.

Finally, SOS [3] is a tool that provides a common programming interface towards different NoSQL systems, to access them in a unified way. The definition of tools is complementary to the goal of this paper, which focuses on design issues.

9. CONCLUSION

In this paper, we have studied the problem of database design for NoSQL datastores. We presented NoAM, a logical approach based on an aggregate-oriented view for application data, an intermediate system-independent data model for NoSQL databases, a language to specify data representations, and implementations that take into account the features of specific systems.

This approach is targeted to the class of applications that can indeed benefit from the usage of NoSQL technologies. Therefore we aimed at supporting the typical requirements of scalability (aggregates are units of distribution), consistency (aggregates are units of transactional consistency), and performance (by allowing data access to specific portions of aggregates).

As suggested by SOS [3], the commonalities of NoSQL datastores can be exploited to provide application developers with a framework that defines a uniform programming interface to access different NoSQL databases, in a transparent way. Currently, we are developing a NoSQL mapping framework with this goal, to support the approach presented in this paper. The application designer identifies the aggregates that should be managed in a NoSQL database. Then the database designer makes use of the language to specify a data representation, in a system-independent way, that the mapping framework will interpret in order to represent the aggregates in the datastore and to handle operations over them. NoSQL database design can benefit from such a tool, as we envision the possibility to use the mapping framework in a flexible way, to tune the choice of the data representation, as well as to select the most suitable target NoSQL datastore.

10. REFERENCES
