Computer Standards & Interfaces 67 (2020) 103149

Contents lists available at ScienceDirect

Computer Standards & Interfaces

journal homepage: www.elsevier.com/locate/csi

Data modeling in the NoSQL world☆

Paolo Atzeni[a], Francesca Bugiotti[b], Luca Cabibbo[a,*], Riccardo Torlone[a]

[a] *Università Roma Tre, Italy*
[b] *Laboratoire de Recherche en Informatique (LRI) CentraleSupélec, Université Paris-Saclay, France*

A R T I C L E I N F O

Keywords:
Data models
Database design
NoSQL systems

A B S T R A C T

NoSQL systems have gained their popularity for many reasons, including the flexibility they provide in organizing data, as they relax the rigidity provided by the relational model and by the other structured models. This flexibility and the heterogeneity that has emerged in the area have led to a little use of traditional modeling techniques, as opposed to what has happened with databases for decades.

In this paper, we argue how traditional notions related to data modeling can be useful in this context as well. Specifically, we propose NoAM (NoSQL Abstract Model), a novel abstract data model for NoSQL databases, which exploits the commonalities of various NoSQL systems. We also propose a database design methodology for NoSQL systems based on NoAM, with initial activities that are independent of the specific target system. NoAM is used to specify a system-independent representation of the application data and, then, this intermediate representation can be implemented in target NoSQL databases, taking into account their specific features. Overall, the methodology aims at supporting scalability, performance, and consistency, as needed by next-generation web applications.

1. Introduction

NoSQL database systems are today an effective solution to manage large data sets distributed over many servers. A primary driver of interest in NoSQL systems is their support for next-generation Web applications, for which relational DBMSs are not well suited. These are simple OLTP applications for which (i) data have a structure that does not fit well in the rigid structure of relational tables, (ii) access to data is based on simple read–write operations, (iii) relevant quality requirements include scalability and performance, as well as a certain level of consistency [2,3].

NoSQL technology is characterized by a high heterogeneity; indeed, more than fifty NoSQL systems exist [4], each with different characteristics. They can be classified into a few main categories [2], including key-value stores, document stores, and extensible record stores. In any case, this heterogeneity is highly problematic to application developers [4], even within each category.

Beside the differences between the various systems, NoSQL datastores exhibit an additional phenomenon: they usually support significant flexibility in data, with limited (if any) use of the notion of schema as it is common in databases. So, the organization of data, and their regularity, is mainly hard-coded within individual applications and is not exposed, probably because there is little need for sharing data between applications. Indeed, the notion of schema, and the need for a separation between data and programs, were motivated in databases by the need for sharing data between applications. If this requirement does not hold any longer, many developers are led to believe that the importance of schemas gets reduced or even disappears.

As the idea of data model is usually tightly related to that of schema, this “schemaless” point view may lead to claim that the very notion of model and of modeling activities becomes irrelevant with respect to NoSQL databases. The goal of this paper is to argue that models and modeling do have an interesting role in this area. Indeed, modeling is an abstraction process, and this helps in general and probably even more in a world of diversity, as the analyst/designer can reason at a high level, before delving into the details of the specific systems. Instead, given the variety of systems, it is currently the case that the design process for NoSQL applications is mainly based on best practices and guidelines [5], which are specifically related to the selected system [6–8], with no systematic methodology. Several authors have observed that the development of high-level methodologies and tools supporting NoSQL database design are needed [9–11], and models here are definitely needed, in order to achieve some level of generality.

Let us recall the various reasons for which modeling is considered important in database design and development [12]. First of all, beside

☆ This paper extends a short article appeared in the Proceedings of the 33rd International Conference on Conceptual Modeling (ER 2014) with the title *Database Design for NoSQL Systems* [1].
* Corresponding author.

https://doi.org/10.1016/j.csi.2016.10.003
Received 25 March 2016; Received in revised form 30 September 2016; Accepted 6 October 2016
Available online 15 October 2016
0920-5489/ © 2016 Elsevier B.V. All rights reserved.

being crucial in the conceptual and logical design phases, it offers support throughout the lifecycle, from requirement analysis, where it helps in giving a structure to the process, to coding and maintenance, where it gives valuable documentation. The main point to be mentioned is that modeling allows the specialist to describe the domain of interest and the application from various perspectives and at various levels of abstraction. Moreover, it provides support to communication (and to individual comprehension). Finally, it provides support to performance management, as physical database design is also based on data structures, and query processing efficiency is often based on reference to the regularity of data.

Conceptual and logical modeling, as they are currently known, were developed in the database world, with specific attention to relational systems, but found applications also in other contexts. Indeed, while the importance of relational databases was clear since the Eighties, it was soon understood that there were many “non-business” application domains for which other modeling features were needed: the advocates of object-oriented databases observed, more or less at the same time, that some requirements were not satisfied, such as those in CAD, CASE, and multimedia and text management [13]. This led to the development of models with nested structures, more complex than the relational one, and less regular, and so more difficult to manage.

Flexibility in structures was also required in another area, which emerged a decade later, and has since been very important: the area of Web applications, where there were at least two kinds of developments concerned with models. On the one hand, work on complex object models for representing hypertexts [14–16], and on the other hand significant development in semistructured data, especially with reference to XML [17].

Another recurring claim in the database world in the last 10 or 15 years has been the fact that, while relational databases are a *de facto* standard, it is not the case that there is one solution that works well for all kinds of applications. As Stonebraker and Çetintemel [18] argued, it is not the case that “one size fits all,” and different engines and technologies are needed in different contexts, for example OLAP and OLTP have different requirements, but the same holds for other kinds of applications, such as stream processing, sensor networks, or scientific databases.

The NoSQL movement emerged for a number of motivations, including most of the above, with the goal of supporting highly scalable systems, with specific requirements, usually with very simple operations over many nodes, on sets of data that have flexible structure. Given that there are many different applications and the specific requirements vary, many systems have emerged, each offering a different way of organizing data and a different programming interface. Heterogeneity can become a problem if migration or integration are needed, as this is often the case, in a world with changing requirements and new technological developments. Also, the availability of many different systems, with different implementations, has led to different design techniques, usually related just to individual systems or small families thereof.

In this paper we argue that a model-based approach can be useful to tackle the difficulties related to heterogeneity, and provide support in the form of abstraction. In fact, modeling can be at the basis of a design process, at various level; at a higher one to represent the features of interest for the application, and at a lower one to describe some implementation features in a concrete but system-independent way.

Indeed, we will present a high-level data model for NoSQL databases, called *NoAM* (NoSQL Abstract Model) and show how it can be used as an intermediate data representation in the context of a general design methodology for NoSQL databases having initial steps that are independent of the individual target system. We propose a design process that includes a conceptual phase, as common in traditional application, followed (and this is unconventional and original) by a system-independent logical design phase, where the intermediate representation is used, as the basis for both modeling and performance aspects, with only a final phase that takes into account the specific features of individual systems.

The rest of the paper is organized as follows. In Section 2, we illustrate the features of the main categories of NoSQL systems arguing that, for each of them, there exists a sort of data model. In Section 3 we present NoAM, our system-independent data model for NoSQL databases, and in Section 4 we discuss our design methodology for NoSQL databases. In Section 5 we briefly review some related literature. Finally, in Section 6 we draw some conclusions.

2. NoSQL data models

In this section we briefly present and compare a number of representative NoSQL systems, to make apparent the heterogeneity (as well as the similarities) in the way they organize data and in their programming interfaces. We first introduce a sample application dataset, and then we show how to represent these data in the representative systems we consider.

2.1. Running example

Let us consider, as a running example, an application for an on-line social game. This is indeed a typical scenario in which the use of a NoSQL database is suitable, that is, a simple next-generation Web

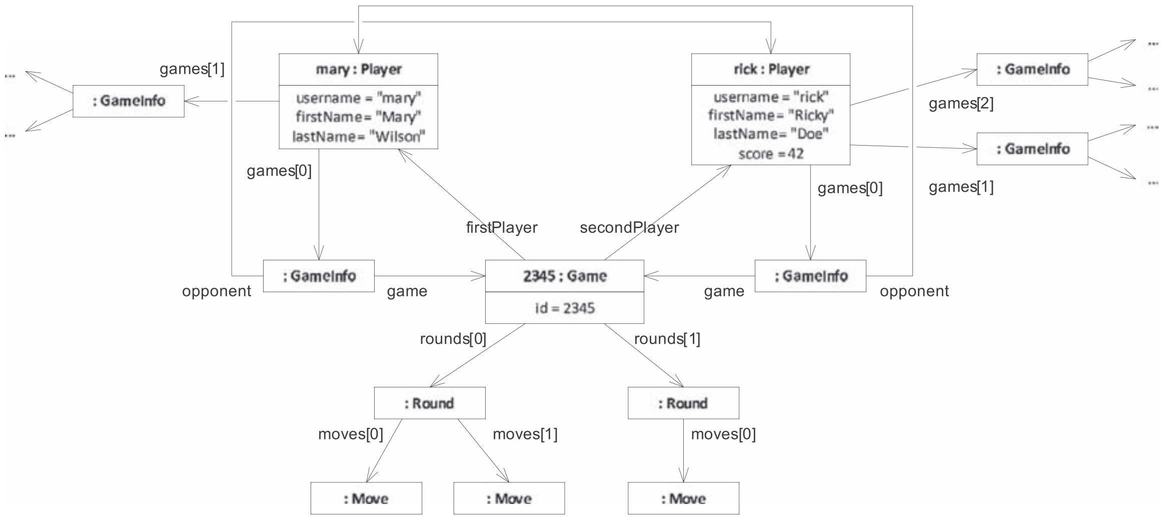


Fig. 1. Sample application objects.

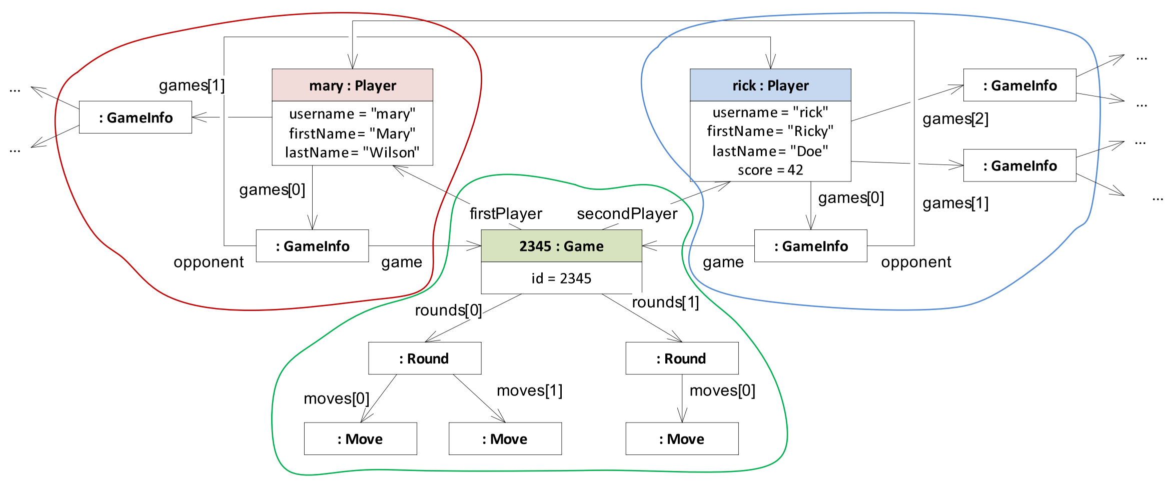


Fig. 2. Sample aggregates (as groups of objects).

application (as discussed in the Introduction).

The application should manage various types of objects, including players, games, and rounds. A few representative objects are shown in Fig. 1. The figure is a UML object diagram. Boxes and arrows denote objects and relationships between them, respectively.

To represent a dataset in a NoSQL database, it is often useful to arrange data in aggregates [19,20]. Each *aggregate* is a group of related application objects, representing a unit of data access and atomic manipulation. In our example, relevant aggregates are players and games, as shown by closed curves in Fig. 2. Note that the rounds of a game are grouped within the game itself. In general, aggregates can be considered as complex-value objects [21], as shown in Fig. 3.

The data access operations needed by our on-line social game are simple read-write operations on individual aggregates; for example, create a new player and retrieve a certain game. Other operations involve just a portion of an aggregate; for example, add a round to an existing game. In general, it is indeed the case that most real applications require only operations that access individual aggregates [2,22].

2.2. NoSQL database models

NoSQL database systems organize their data according to quite different data models. They usually provide simple read-write data-access operations, which also differ from system to system.

Despite this heterogeneity, a few main categories can be identified according to the modeling features of these systems [2,3]: key-value stores, document stores, extensible record stores, plus others (e.g., graph databases) that are beyond the scope of this paper.

2.3. Key-value stores

In general, in a *key-value store*, a database is a schemaless collection of key-value pairs, with data access operations on either individual key-value pairs or groups of related pairs.

As a representative key-value store we consider here *Oracle NoSQL* [23]. In this system, *keys* are structured; they are composed of a *major key* and a *minor key*. The major key is a non-empty sequence of strings. The minor key is a sequence of strings. Each element of a key is called a *component* of the key. On the other hand, each *value* is an uninterpreted binary string.

A sample key-value is the pair composed of key */Player/mary/-/username* and value "*mary*. In the key, symbol '/' separates key components, while symbol '-' separates the major key from the minor key. The distinction between major key and minor is especially relevant to control data distribution and sharding.

Player:mary : ⟨
username : *"mary"*,
firstName : *"Mary"*,
lastName : *"Wilson"*,
games : {
⟨ *game* : Game:2345, *opponent* : Player:rick ⟩,
⟨ *game* : Game:2611, *opponent* : Player:ann ⟩
}
⟩

Player:rick : ⟨
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:2345 : ⟨
id : *"2345"*,
firstPlayer : Player:mary,
secondPlayer : Player:rick,
rounds : {
⟨ *moves* : . . . , *comments* : . . . ⟩,
⟨ *moves* : . . . , *actions* : . . . , *spell* : . . . ⟩
}
⟩

Fig. 3. Sample aggregates (as complex values).

In a pair, the value can be either a simple value (such as the string "*mary*") or a complex value. In the former case, it is common to use some data interchange format (such as XML, JSON, and Protocol Buffers [24]) to represent such complex values.

Oracle NoSQL offers simple atomic access operations, to access and modify individual key-value pairs: put(*key*, *value*) to add or modify a key value pair and get(*key*) to retrieve a value, given the key. Oracle NoSQL also provides an atomic multiGet(*majorKey*) operation to access a group of related key-value pairs, and specifically the pairs having the same major key. Moreover, it offers an execute operation for executing

key (/major/key/-)	*value*
/Player/mary/-	*{ username: "mary", firstName: "Mary", ... }*
/Player/rick/-	*{ username: "rick", firstName: "Ricky", ... }*
/Game/2345/-	*{ id: "2345", firstPlayer: "Player:mary", ... }*

(a) Single key-value pair per aggregate

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*: "Player:rick"}
Player/mary/-/games[1]	{*game*: "Game:2611", *opponent*: "Player:ann"}
Player/rick/-/username	*"rick"*
Player/rick/-/firstName	*"Ricky"*
Player/rick/-/lastName	*"Doe"*
Player/rick/-/score	*42*
Player/rick/-/games[0]	{*game*: "Game:2345", *opponent*: "Player:mary"}
Player/rick/-/games[1]	{*game*: "Game:7425", *opponent*: "Player:ann"}
Player/rick/-/games[2]	{*game*: "Game:1241", *opponent*: "Player:johnny"}
Game/2345/-/id	*2345*
Game/2345/-/firstPlayer	"Player:mary"
Game/2345/-/secondPlayer	"Player:rick"
Game/2345/-/rounds[0]	*{moves: ..., comments: ...}*
Game/2345/-/rounds[1]	*{moves: ..., actions: ..., spell: ...}*

(b) Multiple key-value pairs per aggregate

Fig. 4. Representing aggregates in Oracle NoSQL.

multiple put operations in an atomic and efficient way (provided that the keys specified in these operations all share a same major key).

The data representation for a dataset in a key-value store can be based on aggregates. These are two common representations for aggregates:

- Represent an aggregate using a single key-value pair. The key (major key) is the aggregate identifier. The value is the complex value of the aggregate. See Fig. 4(a).
- Represent an aggregate using multiple key-value pairs. Specifically, the aggregate is split in parts that need to be accessed or modified separately, and each part is represented by a distinct but related key-value pair. The aggregate identifier is used as major key for all these parts, while the minor key identifies the part within the aggregate. See Fig. 4(b).

The data access operations provided by key-value stores usually enable an efficient and atomic data access to aggregates with respect to both data representations. Indeed, all systems support the access to individual key-value pairs (useful in the former case) and most of them (such as Oracle NoSQL) provide also the access to groups of related key-value pairs (required in the latter case).

2.4. Document stores

In a *document store*, a database is a set of documents, each having a complex structure and value.

In this category, a widely used system is *MongoDB* [25]. It is an open-source, document-oriented data store that offers a full-index support on any attribute, a rich document-based query API and Map-Reduce support.

In MongoDB, a *database* comprises one or more collections. Each *collection* is a named group of documents. Each *document* is a structured document, that is, a complex value, a set of attribute-value pairs, which can comprise simple values, lists, and even nested documents. Thus, documents are neither freeform text documents nor Office documents. Documents are schemaless, that is, 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. Values constituting documents can be of the following types: (i) *basic types*, such strings

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

Fig. 5. The JSON representation of the complex value of a sample player object.

numbers, dates, and boolean values; (ii) *arrays*, i.e., ordered sequences of values; and (iii) *documents* (or *objects*): a document is a collection of zero or more key-value pairs, where each key is a plain string, while each value is of any of these types. Fig. 5 shows a JSON representation of the complex value of a sample player aggregate object of Figs. 2 and 3.

A *main document* is a top-level document with a unique identifier, represented by a special attribute *_id*, associated to a value of a special type *ObjectId*.

Data access operations are usually over individual documents, which are units of data distribution and atomic data manipulation. The basic operations offered by MongoDB are as follows: insert(*coll*, *doc*) adds a main document *doc* into collection *coll*; and find(*coll*, *selector*) retrieves from collection *coll* all main documents matching document *selector*. The simplest selector is the empty document {}, which matches with every document; it allows to retrieve all documents in a collection. Another useful selector is document {*_id:ID*}, which matches with the document having identifier *ID*. There is also an operation to update a document. Moreover, it is also possible to access or update just a specific portion of a document.

In a document store, each aggregate is usually represented by a single main document. The document collection corresponds to the aggregate class (or type). The document identifier *ID* is the aggregate identifier. The content of the document is the complex-value of the aggregate, in JSON/BSON, including also an additional key-value pair {*_id:ID*} for the identifier. See Fig. 6.

Also in this case, the data access operations offered by document stores (such as MongoDB) provide an atomic and efficient data access to aggregates. Specifically, they generally support both operations on individual aggregates, or to specific portions of them, thereof.

2.5. Extensible record stores

In an *extensible record store*, a database is a set of tables, each table is a set of rows, and each row contains a set of attributes (or columns), each with a name and a value. Rows in a table are not required to have the same attributes. Data access operations are usually over individual rows, which are units of data distribution and atomic data manipulation.

A representative extensible record store is *Amazon DynamoDB* [26], a NoSQL database service provided on the cloud by Amazon Web Services (AWS). In DynamoDB 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

collection	*document id*	*document*
Player	mary	{"_id":"mary", "username":"mary", "firstName":"Mary", ...}
Player	rick	{"_id":"rick", "username":"rick", "firstName":"Rock", ...}
Game	2345	{"_id":"2345", "firstPlayer":"Player:mary", ...}

Fig. 6. Representing aggregates in MongoDB (abridged).

mostly schemaless.

Specifically, the primary key is composed of a *partition key* and an optional *sort key*. If the primary key of a table includes a sort key, then DynamoDB stores together all the items having the same partition key, in such a way that they can be accessed in an efficient way.

Distribution is operated at the item level and, for each table, is controlled by the partition key only.

Some operations offered by DynamoDB are as follows: putItem(*table*, *key*, *av*) adds (or modifies) a new item in table *table* with primary key *key*, using the set of attribute-value pairs *av*; and getItem(*table*, *key*) retrieves the item of table *table* having primary key *key*. It is also possible to access or update just a subset of the attributes of an item. All these operations can be executed in an efficient way.

In an extensible record store (such as DynamoDB), each aggregate can be represented by a record/row/item. The table corresponds to the aggregate class (or type). The primary key (partition key) is the aggregate identifier. Then, the item can have a distinct attribute-value pair for each top-level attribute of the complex value of the aggregate (or for each major part of the aggregate that needs to be accessed separately). See Fig. 7.

Again, the data access operations provided by the systems in this category support an efficient data access to aggregates or to specific portions of them.

2.6. Comparison

To summarize, it is possible to say that each NoSQL system provides a number of “modeling elements” to organize data, which can be considered the “data model” of the system. Moreover, the various systems can be effectively classified in a few main categories, where each category is based on “data models” that, even though not identical, do share some similarities. In the next section we show that it is possible to pursue these similarities, thus defining an “abstract data model” for NoSQL databases.

3. The NoAM data model

In this section we present *NoAM* (*NoSQL Abstract Data Model*), a system-independent data model for NoSQL databases. In the following section we will also discuss how this data model can be used to support the design of NoSQL databases.

Intuitively, the NoAM data model exploits the commonalities of the data modeling elements available in the various NoSQL systems and introduces abstractions to balance their differences and variations.

A first observation is that all NoSQL systems have a data modeling element that is a data access and distribution unit. By “data access unit” we mean that the system offers operations to access and manipulate an individual unit at a time, in an atomic, efficient, and scalable way. By “distribution unit” we mean that each unit is entirely stored in a server of the cluster, whereas different units are distributed among the various servers. With reference to major NoSQL categories, this element is: (i) a group of related key-value pairs, in key-value stores; (ii) a document, in document stores; or (iii) a record/row/item, in extensible record stores.

In NoAM, a data access and distribution unit is modeled by a *block*. Specifically, a block represents a *maximal* data unit for which atomic, efficient, and scalable access operations are provided. Indeed, while the access to an individual block can be performed in an efficient way in the various systems, the access to multiple blocks can be quite inefficient. In particular, NoSQL systems do not usually provide an efficient “join” operation. Moreover, most NoSQL systems provide atomic operations only over single blocks and do not support the atomic manipulation of a group of blocks. For example, MongoDB [25] provides only atomic operations over individual documents, whereas Bigtable does not support transactions across rows [22].

A second common feature of NoSQL systems is the ability to access and manipulate just a component of a data access unit (i.e., of a block). This component is: (i) an individual key-value pair, in key-value stores; (ii) a field, in document stores; or (iii) a column, in extensible record stores. In NoAM, such a smaller data access unit is called an *entry*.

Finally, most NoSQL databases provide a notion of collection of data access units. For example, a table in extensible record stores or a document collection in document stores. In NoAM, a collection of data access units is called a *collection*.

According to the above observations, the NoAM data model is defined as follows.

- A NoAM *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 its value (either complex or scalar), called the *entry value*.

For example, Fig. 8 shows a possible representation of the aggregates of Figs. 2 and 3 in terms of the NoAM data model. There, outer boxes denote blocks representing aggregates, while inner boxes show entries. Note that entry values can be complex, being this another commonality of various NoSQL systems.

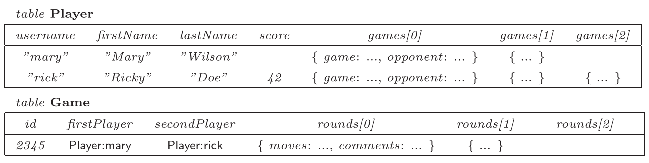


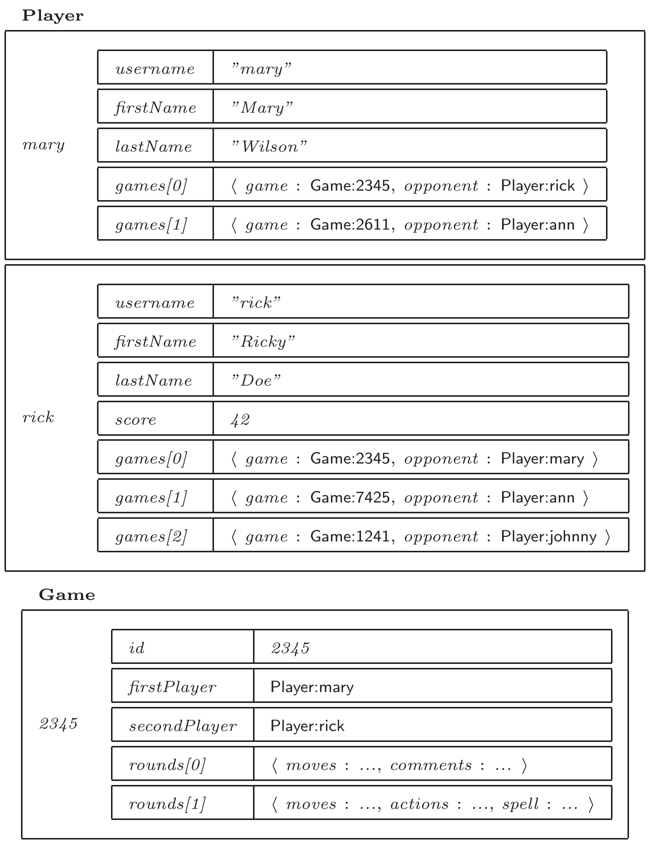
table **Player**

username	*firstName*	*lastName*	*score*	*games[0]*	*games[1]*	*games[2]*
"mary"	*"Mary"*	*"Wilson"*		{ *game*: ..., *opponent*: ... }	{ ... }	
"rick"	*"Ricky"*	*"Doe"*	*42*	{ *game*: ..., *opponent*: ... }	{ ... }	{ ... }

table **Game**

id	*firstPlayer*	*secondPlayer*	*rounds[0]*	*rounds[1]*	*rounds[2]*
2345	Player:mary	Player:rick	{ *moves*: ..., *comments*: ... }	{ ... }	

Fig. 7. Representing aggregates in DynamoDB (abridged).



Player

mary	*username*	*"mary"*
	firstName	*"Mary"*
	lastName	*"Wilson"*
	games[0]	⟨ *game* : Game:2345, *opponent* : Player:rick ⟩
	games[1]	⟨ *game* : Game:2611, *opponent* : Player:ann ⟩
rick	*username*	*"rick"*
	firstName	*"Ricky"*
	lastName	*"Doe"*
	score	*42*
	games[0]	⟨ *game* : Game:2345, *opponent* : Player:mary ⟩
	games[1]	⟨ *game* : Game:7425, *opponent* : Player:ann ⟩
	games[2]	⟨ *game* : Game:1241, *opponent* : Player:johnny ⟩

Game

2345	*id*	*2345*
	firstPlayer	Player:mary
	secondPlayer	Player:rick
	rounds[0]	⟨ *moves* : ..., *comments* : ... ⟩
	rounds[1]	⟨ *moves* : ..., *actions* : ..., *spell* : ... ⟩

Fig. 8. A sample database in NoAM.

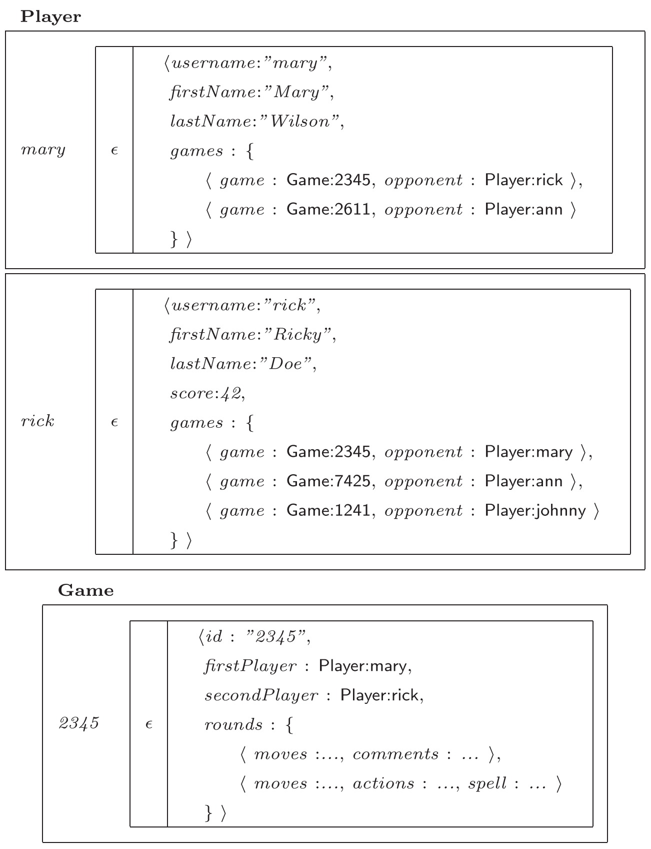

Player
mary
ϵ
⟨username:"mary",
firstName:"Mary",
lastName:"Wilson",
games : {
⟨ game : Game:2345, opponent : Player:rick ⟩,
⟨ game : Game:2611, opponent : Player:ann ⟩
} ⟩
rick
ϵ
⟨username:"rick",
firstName:"Ricky",
lastName:"Doe",
score:42,
games : {
⟨ game : Game:2345, opponent : Player:mary ⟩,
⟨ game : Game:7425, opponent : Player:ann ⟩,
⟨ game : Game:1241, opponent : Player:johnny ⟩
} ⟩
Game
2345
ϵ
⟨id : "2345",
firstPlayer : Player:mary,
secondPlayer : Player:rick,
rounds : {
⟨ moves :..., comments : ... ⟩,
⟨ moves :..., actions : ..., spell : ... ⟩
} ⟩

Fig. 9. Another NoAM sample database.

Note that the same data can be usually represented in different ways. Compare, for example, Fig. 8 with Fig. 9. We will discuss this possibility in the next section.

In summary, NoAM describes in a uniform way the features of many NoSQL systems, and so can be effectively used, as we show in the next section, for an intermediate representation in a NoSQL database design methodology.

4. System-independent design of NoSQL databases with NoAM

The main goal of NoAM is to support a design methodology for NoSQL databases that has initial activities that are independent of the specific target system. In particular, in our approach, NoAM is used to specify an intermediate, system-independent representation of the application data. The implementation in a target NoSQL system is then a final step, with a translation that takes into account its peculiarities.

The motivations to consider database design for NoSQL systems are as follows. It is important to notice that despite the fact that NoSQL databases are claimed to be "schemaless," the data of interest for applications do show some structure, which should be mapped to the modeling elements (collections, tables, documents, key-value pairs) available in the target system. Moreover, different alternatives in the organization of data in a NoSQL database are usually possible, but they are not equivalent in supporting qualities such as performance, scalability, and consistency (which are typically required when a NoSQL database is adopted). For example, a "wrong" database representation can lead to performance that are worse by an order of magnitude as well as to the inability to guarantee atomicity of important operations.

Specifically, our design methodology has the goal of designing a "good" representation of the application data in a target NoSQL database, and is intended to support major qualities such as performance, scalability, and consistency, as needed by next-generation Web applications.

The NoAM approach is based on the following main activities:

- *conceptual data modeling* and *aggregate design*, to identify the various entities and relationships thereof needed in an application, and to group related entities into aggregates;
- *aggregate partitioning* and *high-level NoSQL database design*, where aggregates are partitioned into smaller data elements and then mapped to the NoAM intermediate data model;
- *implementation*, to map the intermediate data representation to the specific modeling elements of a target datastore.

In this approach, only the implementation depends on the target datastore.

We will discuss the various steps of this approach in the rest of this section.

4.1. Conceptual modeling and aggregate design

The methodology starts, as it is usual in database design, by building a conceptual representation of the data of interest, in terms of entities, relationships, and attributes. (This activity is discussed in most database textbooks, e.g., [12].) Following Domain-Driven Design (DDD [19]), which is a widely followed object-oriented methodology, we assume that the outcome of this activity is a conceptual UML class diagram defining the entities, value objects, and relationships of the application. An *entity* is a persistent object that has independent existence and is distinguished by a unique identifier (e.g., a player or a game, in our running example). A *value object* is a persistent object which is mainly characterized by its value, without an own identifier (e.g., a round or a move). Then, the methodology proceeds by identifying aggregates.

The design of aggregates has the goal of identifying the classes of aggregates for an application, and various approaches are possible. After the preliminary conceptual design phase, entities and value objects are grouped into 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.

The relevant decisions in aggregate design involve the choice of aggregates and of their boundaries. This activity can be driven by the data access patterns of the application operations, as well as by scalability and consistency needs [19]. Specifically, aggregates should be designed as the units on which atomicity must be guaranteed [20] (with eventual consistency for update operations spanning multiple aggregates [27]). In general, it is indeed the case that most real applications require only operations that access individual aggregates [2,22]. Each aggregate should be large enough so as to include all the data required by a relevant data access operation. (Note that NoSQL systems do not provide a "join" operation, and this is a main motivation for clustering each group of related application objects into an aggregate.) Furthermore, to support strong consistency (that is, atomicity) of update operations, each aggregate should include all the data involved by some integrity constraints or other forms of business rules [28]. On the other hand, aggregates should be as small as possible; small aggregates reduce concurrency collisions and support performance and scalability requirements [28].

Thus, aggregate design is mainly driven by data access operations. In our running example, the online game application needs to manage various collections of objects, including players, games, and rounds. Fig. 2 shows a few representative application objects. (There, 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.) When a player connects to the application, all data on the player should be retrieved, including an overview of the games she is currently playing. Then, the player can select to continue a game, and data on the selected game should be retrieved. When a player completes a round in a game she is playing, then the game should be updated. These operations suggest that the candidate aggregate classes are players and games. Fig. 2 also shows how application objects can be grouped in aggregates. (There, a closed curve denotes the boundary of an aggregate.)

As we mentioned above, aggregate design is also driven by consistency needs. Assume that the application should enforce a rule specifying that a round can be added to a game only if some condition that involves the other rounds of the game is satisfied. An individual round cannot check, alone, the above condition; therefore, it cannot be an aggregate by itself. On the other hand, the above business rule can be supported by a game (comprising, as an aggregate, its rounds).

In conclusion, the aggregate classes for our sample application are **Player** and **Game**, as shown in Figs. 2 and 3.

4.2. Data representation in NoAM and aggregate partitioning

In our approach, we use the NoAM data model (Section 3) as an intermediate model between application aggregates (Section 4.1) and NoSQL databases (Section 2). We represent each class of aggregates by means of a distinct collection, and each individual aggregate by means of a block. We use the class name to name the collection, and the identifier of the aggregate as block key. The complex value of each aggregate is represented by a set of entries in the corresponding block. For example, the aggregates of Figs. 2 and 3 can be represented by the NoAM database shown in Fig. 8. The representation of aggregates as blocks is motivated by the fact that both concepts represent a unit of data access and distribution, but at different abstraction levels. Indeed, NoSQL systems provide efficient, scalable, and consistent (i.e., atomic) operations on blocks and, in turn, this choice propagates such qualities to operations on aggregates.

In general, an application dataset of aggregates can be represented in NoAM database in several different ways. Each *data representation* for a dataset δ is a NoAM database D_δ representing δ. Specifically, the various data representations for a dataset differ only in the choice of the entries used to represent the complex value of each aggregate. We first discuss basic data representation strategies, which we illustrate with respect to the example described in Fig. 3. We then introduce additional and more flexible data representations.

A simple data representation strategy, called *Entry per Aggregate Object* (*EAO*), represents each individual aggregate using a single entry. The entry key is empty. The entry value is the whole complex value of the aggregate. The data representation of the aggregates of Fig. 3 according to the EAO strategy is shown in Fig. 9.

Another data representation strategy, called *Entry per Top-level Field* (*ETF*), represents each aggregate by means of multiple entries, using a distinct entry for each top-level field of the complex value of the aggregate. For each top-level field f of an aggregate o, it employs an entry having as value the value of field f in the complex value of o (with values that can be complex themselves), and as key the field name f. Fig. 10 shows the data representation of the aggregates of Fig. 3 according to the ETF strategy.

As a comparison, we can observe that the EAO data representation uses a block with a single entry to represent the **Player** object having username *mary*, while the ETF representation needs a block with four entries, corresponding to fields *username*, *firstName*, *lastName*, and *games*. Moreover, blocks in EAO do not depend on the structure of aggregates, while blocks in ETF depend on the top-level structure of aggregates (which can be "almost fixed" within each class).

The general data representation strategies we just described can be suited in some cases, but they are often too rigid and limiting. For example, none of the above strategies leads to the data representation shown in Fig. 8. The main limitation of such general data representa-

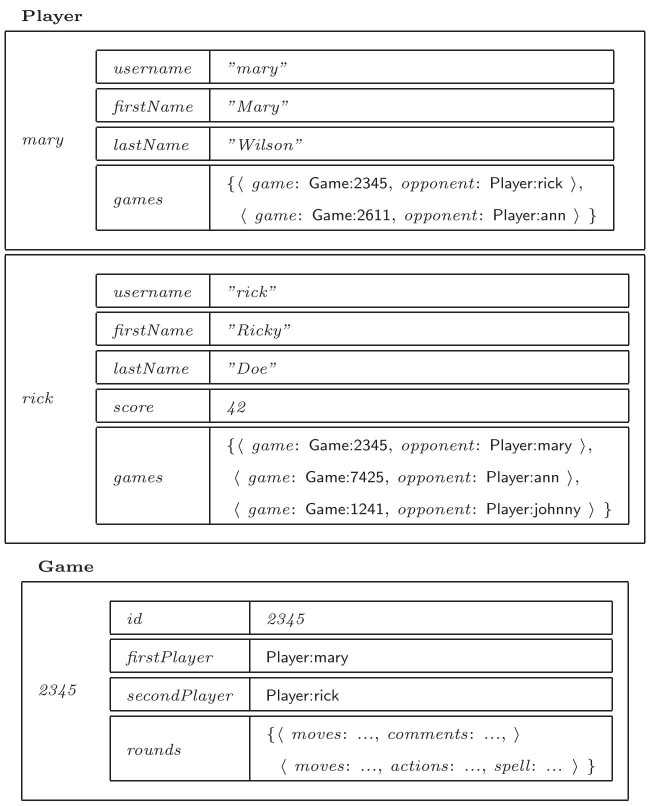

Player
mary
username | "mary"
firstName | "Mary"
lastName | "Wilson"
games | {⟨ game: Game:2345, opponent: Player:rick ⟩, ⟨ game: Game:2611, opponent: Player:ann ⟩ }
rick
username | "rick"
firstName | "Ricky"
lastName | "Doe"
score | 42
games | {⟨ game: Game:2345, opponent: Player:mary ⟩, ⟨ game: Game:7425, opponent: Player:ann ⟩, ⟨ game: Game:1241, opponent: Player:johnny ⟩ }
Game
2345
id | 2345
firstPlayer | Player:mary
secondPlayer | Player:rick
rounds | {⟨ moves: ..., comments: ..., ⟩ ⟨ moves: ..., actions: ..., spell: ... ⟩ }

Fig. 10. The ETF data representation.

tions is that they refer only to the structure of aggregates, and do not take into account the data access patterns of the application operations. Therefore, these strategies are not usually able to support the performance of these operations. This motivates the introduction of aggregate partitioning.

We first need to introduce a preliminary notion of *access path*, to specify a "location" in the structure of a complex value. Intuitively, if v is a complex value and w is a value (possibly complex as well) occurring in v, then the access path *ap* for w in v represents the sequence of "steps" that should be taken to reach the component value w in v. More precisely, an access path *ap* is a (possibly empty) sequence of *access steps*, $ap = p_1\, p_2 \ldots p_n$, where each step p_i identifies a component value in a structured value. Furthermore, if v is a complex value and *ap* is an access path, then $ap(v)$ denotes the component value identified by *ap* in v.

For example, consider the complex value v_{mary} of the **Player** aggregate having username *mary* shown in Fig. 3. Examples of access paths for this complex value are *firstName* and *games*[0].*opponent*. If we apply these access paths to v_{mary}, we access values *Mary* and Player:rick, respectively.

A complex value v can be represented using a set of entries, whose keys are access paths for v. 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). Specifically, in NoAM we represent each aggregate by means of a *partition* of its complex value v, that is, a set E of entries that fully cover v, without redundancy. Consider again the complex value v_{mary} shown in Fig. 3; a possible entry for v_{mary} is the pair ⟨*games* [0]. *opponent*, Player:rick⟩. We have already applied the above intuition earlier in this section. For example, the ETF data representation (shown in Fig. 10) uses field names as entry keys (which are indeed a case of access paths) and field values as entry values.

Aggregate partitioning can be based on the following guidelines (which are a variant of guidelines proposed in [12] in the context of logical database design):

- 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.

The application of the above guidelines suggests a partitioning of aggregates, which we will use to guide the representation in the target database.

For example, in our sample application, consider the operations involving games and rounds. When a player selects to continue a game, data on the selected game should be retrieved. When a player completes a round in a game she is playing, then the aggregate for the game should be updated. To support performance, it is desirable that this update is implemented in the database just as an addition of a round to a game, rather than a complete rewrite of the whole game. Thus, data for each individual round is always read or written together. Moreover, data for the various rounds of a game are read together, but each round is written separately. Therefore, each round is a candidate to be represented by an autonomous entry. These observations lead to a data representation for games shown in Fig. 8. However, apart from rounds, the remaining data for each game comprises just a few fields, which can be therefore represented together in a single entry. This further observation leads to an alternative data representation for games, shown in Fig. 11.

4.3. Implementation

We now discuss how a NoAM data representation can be implemented in a target NoSQL database. Given that NoAM generalizes the features of the various NoSQL systems, while keeping their major aspects, it is rather straightforward to perform this activity. We have implementations for various NoSQL systems, including Cassandra, Couchbase, Amazon DynamoDB, HBase, MongoDB, Oracle NoSQL, and Redis. For the sake of space, we discuss the implementation only with respect to a single representative system for each main NoSQL category. Moreover, with reference to the same aggregate objects of Figs. 2 and 3 we will sometimes show only the data for one aggregate. Similar representations can be obtained for the other aggregates of the running example.

4.3.1. Key-value store: Oracle NoSQL

In the key-value store Oracle NoSQL [23] (Section 2.3), a data representation D for an application dataset can be implemented as follows. We use a key-value pair for each entry $\langle ek, ev\rangle$ 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 ek (recall that ek is an access path, which we represent using a distinct key component for each of its steps). An example of key is */Player/mary/-/firstName*, 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 ev; for example, *Mary*. The value can be either simple or a serialization of a complex value, e.g., in JSON.

The retrieval of a block can be implemented, in an efficient and atomic way, using a single multiGet operation—this is possible because all the entries of a block share the same major key. The storage of a block can be implemented using various put operations. These multiple put operations can be executed in an atomic way—since, again, all the entries of a block share the same major key.

For example, Fig. 4(b) shows the implementation in Oracle NoSQL of the data representation of Fig. 8. Moreover, Fig. 4(a) shows the implementation in Oracle NoSQL of the EAO data representation of Fig. 9.

An implementation can be considered *effective* if aggregates are indeed turned into units of data access and distribution. The effectiveness of our implementation is based on the use we make of Oracle NoSQL keys, where 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).

More precisely, a technical precaution is needed to guarantee atomic consistency when the selected data representation uses more than one entry per 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 NoSQL system 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 Oracle NoSQL 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 always including in each update operation the access to the entry that includes the identifier of the aggregate (or some other distinguished entry of the block).

4.3.2. Extensible record store: DynamoDB

In the extensible record store Amazon DynamoDB ([26], Section 2.5), the implementation of a NoAM database can be 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.

A NoAM data representation D can be represented in DynamoDB as follows. Consider a block b in a collection C having block key id. According to D, one or multiple entries are used within each block. We 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 DynamoBD table name; (ii) the block key id is used as a DynamoBD primary key in that table; (iii) the set of entries (key-value pairs) of a block b 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. 7 shows the implementation of the NoAM database of Fig. 8.

The retrieval of a block, given its collection C and block key id, can be implemented by performing a single getItem operation, which retrieves the item that contains all the entries of the block. The storage of a block can be implemented using a putItem operation, to save all the entries of the block, in an atomic way. It is worth noting that, using operation getItem, it is also possible to retrieve a subset of the entries of a block. Similarly, using operation updateItem, it is also possible to update just a subset of the entries of a block, in an atomic way.

This implementation is also effective, since DynamoDB controls distribution and atomicity with reference to items.

4.3.3. Document store: MongoDB

In *MongoDB* ([29], Section 2.4), which is a document store, a natural implementation for a NoAM database can be 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 can be

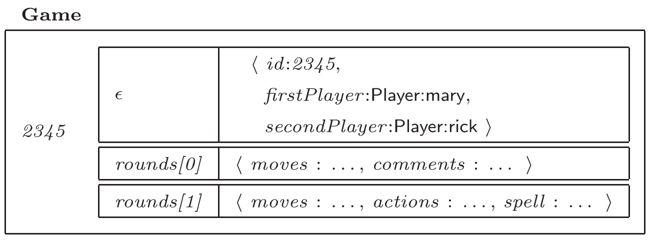

Game
2345
ϵ
⟨ id:2345,
firstPlayer:Player:mary,
secondPlayer:Player:rick ⟩
rounds[0]
⟨ moves : ..., comments : ... ⟩
rounds[1]
⟨ moves : ..., actions : ..., spell : ... ⟩

Fig. 11. An alternative data representation for games (ROUNDS).

defined as a suitable JSON/BSON serialization of the complex value of the entries in *b*, plus a special field to store the block key *id* of *b*, as required by MongoDB, {*_id:id*}.

With reference to a NoAM data representation *D*, consider a block *b* in a collection *C* having block key *id*. If *b* contains just an entry *e*, then the document for *b* is just a serialization of *e*. Otherwise, if *b* contains multiple entries, we 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 in the block *b* is used to fill a (possibly nested) field of document *d*. See Fig. 12.

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.

An alternative implementation for MongoDB is as follows. Each block *b* is represented, again, as a main document for *b*, but using a distinct top-level field-value pair for each entry in the NoAM data 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. 8 leads to the result shown in Fig. 13.

collection **Player**

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

Fig. 12. Implementation in MongoDB.

collection **Player**

id	*document*
mary	{ *_id:"mary"*, *username:"mary"*, *firstName:"Mary"*, *lastName:"Wilson"*, *games[0]:* { *game:"Game:2345"*, *opponent:"Player:rick"* }, *games[1]:* { *game:"Game:2611"*, *opponent:"Player:ann"* } }

Fig. 13. Alternative implementation in MongoDB.

4.4. Experiments

We will now discuss a case study of NoSQL database design, with reference to our running example. For the sake of simplicity, we just focus on the representation and management of aggregates for games.

Data for each game include a few scalar fields and a collection of rounds. The 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.

Assume that, to manage games, we have chosen a key-value store as the target system. The candidate data representations are: (i) using a single entry for each game (as shown in Fig. 9, in the following called EAO); (ii) splitting the data for each game in a group of entries, one for each round, and including all the remaining scalar fields in a separate entry (as shown in Fig. 11, called ROUNDS).

We expect that the first operation (retrieval of a game) performs better in EAO, since it needs to read just a key-value pair, while the second one (addition of a round to a game) is favored by ROUNDS, which does not require to rewrite the whole game.

We ran a number of experiments to compare the above data representations in situations of different application workloads. Each game has, on average, a dozen rounds, for a total of about 8 kB per game. At each run, we simulated the following workloads: (a) game retrievals only (in random order); (b) round additions only (to random games); and (c) a mixed workload, with game retrieval and round addition operations, with a read/write ratio of 50/50. We ran the experiments using different database sizes, and measured the running time required by the workloads. The target system was Oracle NoSQL, deployed over Amazon AWS on a cluster of four EC2 servers.[1]

The results are shown in Fig. 14. Database sizes are in gigabytes, timings are in milliseconds, and points denote the average running time of a single operation. The experiments confirm the intuition that the retrieval of games Fig. 14(a) is always favored by the EAO data representation, for any database size. On the other hand, the addition of a round to an existing game Fig. 14(b) is favored by the ROUNDS data representation. Finally, the experiments over the mixed workload Fig. 14(c) show a general advantage of ROUNDS over EAO, which however decreases as the database size increases. Overall, it turns out that the ROUNDS data representation is preferable.

We also performed other experiments on a data representation that does not conform to the design guidelines proposed in this paper. Specifically, a data representation that divides the rounds of a game into independent key-value pairs, rather than keeping them together in a same block, as suggested by our approach. In this case, the performance of the various operations worsens by at least an order of magnitude. Moreover, with this data representation it is not possible to update a game in an atomic way.

Overall, these experiments show that: (i) the design of NoSQL databases should be done with care as it affects considerably the performance and consistency of data access operations, and (ii) our methodology provides an effective tool for choosing among different alternatives.

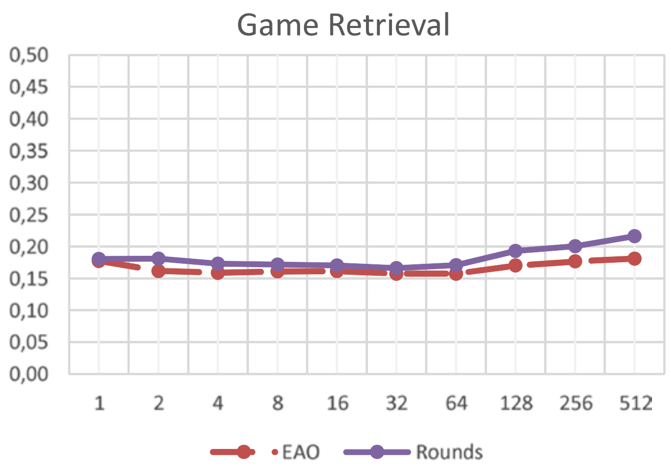
5. Related works

Although several authors have observed that there is a need for data-model approaches to the design and management of NoSQL databases [9–11], very few works have addressed this issue, especially from a general and system-independent point of view. Indeed, most of them propose a solution to a specific problem in a limited scenario.

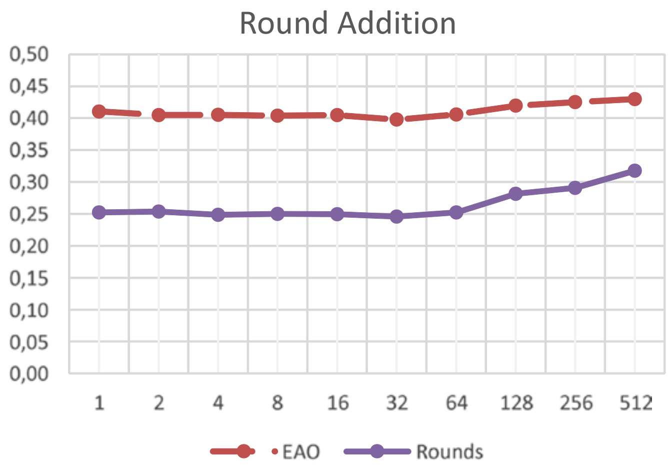
For instance, Pasqualin et al. have recently shown how a document-oriented model can be efficiently implemented in a NoSQL document store [30]. Similarly, Olivera et al. [31] and de Lima and Mello [32] have proposed a data-model based methodology for the design of NoSQL document database [32], whereas Chevalier et al. have addressed the specific problem of leveraging on a document-oriented model for implementing a multidimensional database in a NoSQL document store [33] and in a column-oriented NoSQL database [34].

Most of the other contributions to data modeling for NoSQL systems come from on-line papers, usually published in blogs of

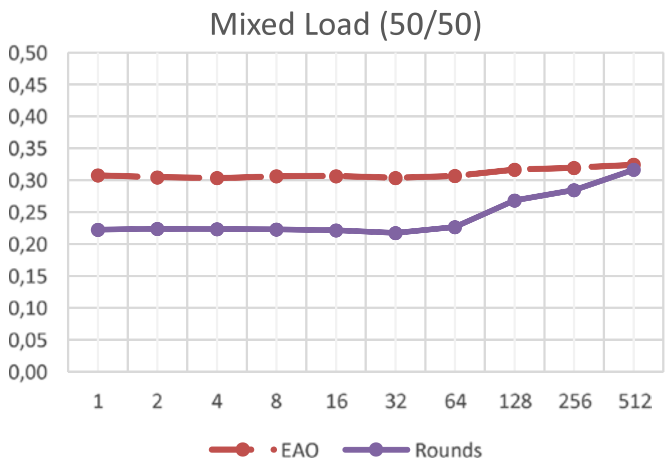
[1] This activity was supported by AWS in Education Grant award.



(a) Game Retrieval



(b) Round Addition



(c) Mixed Load

Fig. 14. Experimental results.

practitioners, that discuss best practices and guidelines for modeling NoSQL databases, most of which are suited only for specific systems. For instance, [5] lists some techniques for implementing and managing data stored in different types of NoSQL systems, while [35] discusses design issues for the specific case of key-value datastores. Similarly, Mior et al. [36] have recently proposed an approach to the problem of schema design for the specific class of extensible record stores. On the system-oriented side, [6–8] illustrate design principles for the specific cases of HBase, MongoDB, and Cassandra, respectively. However, none of them tackles the problem from a general perspective, as we advocate in this paper.

Recently, Ruiz et al. have proposed a reverse engineering strategy aimed at inferring the implicit schema of NoSQL databases [37]. This approach supports the idea that, even in this context, a model-based description of the organization of data is very useful during the entire life-cycle of a data set.

To the best of our knowledge, this paper presents the first general design methodology for NoSQL systems with initial activities that are independent of the specific target system. Our approach to data modeling is based on data aggregates, a notion that is central in NoSQL databases where application data are grouped in atomic units that are accessed and manipulated together [3]. The notion of aggregate also occurs in other contexts with a similar meaning. For example, in Domain Driven Design [19], a widely followed object-oriented software development approach, an aggregate is a group of related application objects, used to govern transactions and distribution. Also Helland [20] advocates the use of aggregates (there called entities) as units of distribution and consistency. In this framework, Baker et al. [38] propose the notion of entity groups, a set of entities that can be manipulated in an atomic way. They also describe a specific mapping of entity groups to Bigtable [22], which however makes the approach targeted only to a specific NoSQL system. Our approach is based on a more abstract database model, NoAM, and is system independent, as it is targeted to a wide class of NoSQL systems.

The issue of identifying data access units in database design shows some similarities with problems studied in the past, such as: (i) the early works on vertical partitioning and clustering [39], 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 [40], where various alternatives obviously exist, with tables that can be very small and handle individual edges, or very wide and handle entire paths, and many alternatives in between.

A major observation from [9] is 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. We have addressed this issue by proposing NoAM, an abstract data model that makes it possible to devise an initial phase of the design process that is independent of any specific system but suitable for each.

Along this line, SOS [41] is a tool that provides a common programming interface towards different NoSQL systems, to access them in a unified way. The interface is based on a simple, high-level common data model which is inspired by those of non-relational systems and provides simple operations for inserting, deleting, and retrieving database objects. However, the definition of tools for data access is complementary to data models and design issues.

Finally, Jain et al. discuss the potential mismatch between the requirements of scientific data analysis and the models and languages of relational database systems [42], whereas Alagiannis et al. [43] advocate a new database design philosophy for emerging applications. This paper tries to provide a contribution to these problems.

6. Conclusion

In this paper we have argued how data modeling can be useful in the NoSQL arena. Specifically, we have proposed a comprehensive methodology for the design of NoSQL databases, which relies on an aggregate-oriented view of application data, an intermediate system-independent data model for NoSQL datastores, and finally an implementation activity that takes into account the features of specific systems.

References

[1] F. Bugiotti, L. Cabibbo, P. Atzeni, R. Torlone, Database design for NoSQL systems, in: Conceptual Modeling—33rd International Conference, ER 2014, Atlanta, GA, USA, October 27–29, 2014. Proceedings, 2014, pp. 223–231.
[2] R. Cattell, Scalable SQL and NoSQL data stores, SIGMOD Record 39 (4) (2010) 12–27.
[3] P.J. Sadalage, M.J. Fowler, NoSQL Distilled, Addison-Wesley, Upper Saddle River, NJ, USA, 2012.
[4] M. Stonebraker, Stonebraker on NoSQL and enterprises, Commun. ACM 54 (8) (2011) 10–11.
[5] I. Katsov, NoSQL Data Modeling Techniques, Highly Scalable Blog, ⟨https://highlyscalable.wordpress.com/2012/03/01/nosql-data-modeling-techniques/⟩, 2012 (accessed February 2016).
[6] A. Khurana, Introduction to HBase Schema Design, ;login. Usenix Mag. 37 (5) (2012) 29–36.
[7] 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 (accessed February 2016).
[8] A. Chebotko, A. Kashlev, S. Lu, A big data modeling methodology for Apache Cassandra, in: IEEE International Congress on Big Data, 2015, pp. 238–245.
[9] P. Atzeni, C.S. Jensen, G. Orsi, S. Ram, L. Tanca, R. Torlone, The relational model is dead, SQL is dead, and I don't feel so good myself, SIGMOD Record 42 (2) (2013) 64–68.
[10] A. Badia, D. Lemire, A call to arms: revisiting database design, SIGMOD Record 40 (3) (2011) 61–69.
[11] C. Mohan, History repeats itself: sensible and NonsenSQL aspects of the NoSQL hoopla, in: EDBT, 2013, pp. 11–16.
[12] C. Batini, S. Ceri, S.B. Navathe, Conceptual Database Design: An Entity-Relationship Approach, Benjamin/Cummings, Redwood City, CA, USA, 1992.
[13] F. Bancilhon, Object-oriented database systems, in: Proceedings of the Seventh ACMSIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, March 21–23, 1988, Austin, TX, USA, 1988, pp. 152–162.
[14] P. Atzeni, P. Merialdo, G. Mecca, Data-intensive web sites: design and maintenance, World Wide Web 4 (1–2) (2001) 21–47.
[15] S. Ceri, P. Fraternali, A. Bongio, M. Brambilla, S. Comai, M. Matera, Designing Data-Intensive Web Applications, Morgan Kaufmann, San Francisco, CA, USA, 2003.
[16] G. Mecca, A.O. Mendelzon, P. Merialdo, Efficient queries over web views, IEEE Trans. Knowl. Data Eng. 14 (6) (2002) 1280–1298.
[17] S. Abiteboul, P. Buneman, D. Suciu, Data on the Web: From Relations to Semistructured Data and XML, Morgan Kaufmann, San Francisco, CA, USA, 1999.
[18] M. Stonebraker, U. Çetintemel, “one size fits all”: an idea whose time has come and gone (abstract), in: Proceedings of the 21st International Conference on Data Engineering, ICDE 2005, 5–8 April 2005, Tokyo, Japan, 2005, pp. 2–11.
[19] E. Evans, Domain-Driven Design, Addison-Wesley, Boston, MA, USA, 2003.
[20] P. Helland, Life beyond distributed transactions: an Apostate's opinion, in: CIDR 2007, 2007, pp. 132–141.
[21] S. Abiteboul, R. Hull, V. Vianu, Foundations of Databases, Addison-Wesley, Reading, MA, USA, 1995.
[22] F. Chang, et al., Bigtable: a distributed storage system for structured data, ACM Trans. Comput. Syst. 26 (2) (2008).
[23] Oracle, Oracle NoSQL Database, ⟨http://www.oracle.com/us/products/database/nosql/⟩ (accessed February 2016).
[24] J. Shute, et al., F1: a distributed SQL database that scales, PVLDB 6 (11) (2013) 1068–1079.
[25] MongoDB Inc., MongoDB, ⟨http://www.mongodb.org⟩ (accessed February 2016).
[26] Amazon Web Services, DynamoDB, ⟨http://aws.amazon.com/it/dynamodb⟩ (accessed February 2016).
[27] D. Pritchett, ASE: an ACID alternative, ACM Queue 6 (3) (2008) 48–55.
[28] V. Vernon, Implementing Domain-Driven Design, Addison-Wesley, Upper Saddle River, NJ, USA, 2013.
[29] K. Chodorow, MongoDB: The Definitive Guide, (Eds.), O'Reilly Media, Sebastopol, CA, USA, 2013.
[30] D. Pasqualin, G. Souza, E.L. Buratti, E.C. de Almeida, M.D. Del Fabro, D. Weingaertner, A case study of the aggregation query model in read-mostly NoSQL document stores, in: 20th International Database Engineering & Applications Symposium (IDEAS '16), IDEAS '16, ACM, New York, NY, USA, 2016, pp. 224–229.
[31] H. V. Olivera, M. Holanda, V. Guimarâes, F. Hondo, W. Boaventura, Data modeling for NoSQL document-oriented databases, in: 2nd Annual International Symposium on Information Management and Big Data (SIMBig 2015), vol. 1478 of CEUR Workshop Proceedings, 2015, pp. 129–135.
[32] C. de Lima, R. dos Santos Mello, A workload-driven logical design approach for NoSQL document databases, in: 17th International Conference on Information Integration and Web-based Applications & Services (iiWAS '15), iiWAS '15, ACM, New York, NY, USA, 2015, pp. 73:1–73:10.
[33] M. Chevalier, M.E. Malki, A. Kopliku, O. Teste, R. Tournier, Implementation of multidimensional databases with document-oriented NoSQL, in: 17th International Conference on Big Data Analytics and Knowledge Discovery, (DaWaK 2015), Cham, Switzerland, Lecture Notes in Computer Science, vol. 9263, Springer, 2015, pp. 379–390.
[34] M. Chevalier, M.E. Malki, A. Kopliku, O. Teste, R. Tournier, Implementation of multidimensional databases in column-oriented NoSQL systems, in: 19th East European Conference on Advances in Databases and Information Systems (ADBIS 2015), 2015, pp. 79–91.
[35] T. Olier, Database design using key-value tables, ⟨http://www.devshed.com/c/a/mysql/database-design-using-key-value-tables/⟩, 2006 (accessed February 2016).
[36] M.J. Mior, K. Salem, A. Aboulnaga, R. Liu, Nose: schema design for NoSQL applications, in: 32nd IEEE International Conference on Data Engineering, ICDE 2016, Helsinki, Finland, May 16–20, 2016, pp. 181–192.
[37] D.S. Ruiz, S.F. Morales, J.G. Molina, Inferring versioned schemas from NoSQL databases and its applications, in: 34th International Conference on Conceptual Modeling (ER 2015), 2015, pp. 467–480.
[38] J. Baker, et al., Megastore: Providing scalable, highly available storage for interactive services, in: CIDR 2011, 2011, pp. 223–234.
[39] T.J. Teorey, J.P. Fry, The logical record access approach to database design, ACM Comput. Surv. 12 (2) (1980) 179–211.
[40] D. Florescu, D. Kossmann, Storing and querying XML data using an RDMBS, IEEE Data Eng. Bull. 22 (3) (1999) 27–34.
[41] P. Atzeni, F. Bugiotti, L. Rossi, Uniform access to NoSQL systems, Inf. Syst. 43 (2014) 117–133.
[42] S. Jain, D. Moritz, D. Halperin, B. Howe, E. Lazowska, SQLShare: results from a multi-year SQL-as-a-service experiment, in: Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, 26 June–1 July 2016, pp. 281–293.
[43] I. Alagiannis, R. Borovica-Gajic, M. Branco, S. Idreos, A. Ailamaki, NoDB: efficient query execution on raw data files, Commun. ACM 58 (12) (2015) 112–121.