How I Learned to Stop Worrying and Love NoSQL Databases

Francesca Bugiotti, Luca Cabibbo, Paolo Atzeni, Riccardo Torlone

How I Learned to Stop Worrying and Love NoSQL Databases*

Francesca Bugiotti[1], Luca Cabibbo[2], Paolo Atzeni[2], and Riccardo Torlone[2]
[1]CentraleSupélec & INRIA & Université Paris-Sud and [2]Università Roma Tre

Abstract. The absence of a schema in NoSQL databases can disorient traditional database specialists and can make the design activity in this context a leap of faith. However, we show in this paper that an effective design methodology for NoSQL systems supporting the scalability, performance, and consistency of next-generation Web applications can be indeed devised. The approach is based on NoAM (NoSQL Abstract Model), a novel abstract data model for NoSQL databases, which is used to specify a system-independent representation of the application data. This intermediate representation can be then implemented in target NoSQL databases, taking into account their specific features.

1 Motivation

A common myth on NoSQL databases is that they do not require database design, since they are "schemaless." You can put any data in a NoSQL database, in any way you like, and then retrieve that data. While this is true in some cases (for example, to cache session data in a web application), most often the data of interest for an application do show some structure, which it may be useful to take advantage of.

To uphold the importance of NoSQL database design, we start by discussing an example: an application for an on-line social game. This is a typical scenario in which the use of a NoSQL database is suitable. The application should manage various types of objects, including players, games, and rounds. A few representative objects are shown in Fig. 1. (There, boxes and arrows denote objects and relationships between them, respectively; please ignore, for now, colors and closed curves.)

Assume that we have chosen a key-value store as target NoSQL database. What key-value pairs should we use? A distinct key-value pair for each different *object*? A distinct key-value pair for each different *attribute* of an object? Or should we use each key-value pair to represent a *group of related attributes* of an objects? Or to represent a *group of related objects*? In the latter cases, what are the grouping criteria?

In general, an application dataset can be represented in a NoSQL database in multiple, alternative ways. Thus decisions on the organization of data are required, in any case. We now show that these decisions are significant, as they affect important application qualities such as performance, scalability, and consistency.

Assume that, in the example above, we have chosen the following data representation: *a key-value pair for each application object*. Thus, we have a distinct key-value pair for each player, game, and round, among others (Fig. 2). Then, consider the following operations: (i) add a round to a game, and (ii) retrieve a game (with all its rounds).

*This paper is a short version of [7]. Part of this work was performed while the first author was with Università Roma Tre.

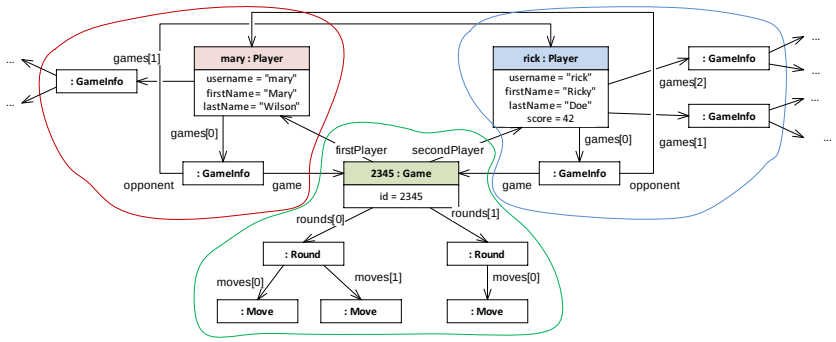

Fig. 1. Sample application objects

Assume also 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.

The proposed representation is unable to guarantee the atomicity of operation (i). Most key-value stores can only support atomicity of operations involving a single key-value pair, whereas operation (i) requires, in the proposed representation, the access to many key-value pairs, namely, a pair for the game and several pairs for its rounds.

Furthermore, the proposed representation is inefficient with respect to operation (ii). Indeed, the retrieval of a game with its rounds requires to "get" multiple key-value pairs, which can be sharded over the various nodes managing the NoSQL datastore, and this implementation is less efficient than the execution of a single "multi-get" access, which is however possible only for key-value pairs localized on a single node.

Scalability is also affected negatively by this representation. Indeed, horizontal scalability can be obtained only if each operation execution can be managed by a single node, whereas in the proposed data representation both operations (i) and (ii) require accessing multiple key-value pairs residing on different shards.

key (*/major/key/-/minor/key*)	*value*
Player/mary/-/	{*username:"mary"*, . . . , *games*: [**"Game:2345"**, . . .]}
Player/rick/-/	{*username:"rick"*, . . . , *games*: [**"Game:2345"**, . . .]}
Game/2345/-/	{*id:"2345"*, *rounds*: [**"Round:2345:0"**, **"Round:2345:1"**, . . .]}
Round/2345/0/-/	. . .
Round/2345/1/-/	. . .

Fig. 2. A "key-value pair for each application object" database representation of the sample objects of Fig. 1 (abridged)

The above example shows that a "wrong" database representation can lead to the inability to guarantee atomicity of important operations and to poor performance and scalability. (We have performed several experiments demonstrating these facts.)

In the remainder of this paper we describe a general design methodology for NoSQL databases aiming at identifying a "good" database representation with respect to atomicity, performance, and scalability. This goal is pursued by referring to models adopted in preliminary steps, which are motivated by the need to properly implement the desired semantics of the application.

2 Overview

Let us consider, as a running example, an application for an on-line social game, introduced in Sect. 1.

The methodology starts by building a conceptual representation of the data of interest, as shown by the sample objects in Fig. 1. We assume, as it is often the case, that this would be done in the interest of the quality of the application being developed.

The methodology proceeds by identifying aggregates. Each *aggregate* is a group of related application objects that should be accessed and/or manipulated together. This activity is relevant to support scalability and consistency, as aggregates provide a natural unit for sharding and atomic manipulation of data. In our example, natural aggregates are players and games, as shown by closed curves in Fig. 1. Note that the rounds of a game are nested within the game itself. In general, aggregates can be considered as complex-value objects [1], as shown in Fig. 3.

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

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

Fig. 3. Sample aggregates (as complex values)

The next activity consists in partitioning aggregates, if needed, into smaller elements to better support the performance of certain operations. Consider for instance a player that completes a round in a game she is playing. In order to update the underlying database, there would be two alternatives: (i) the addition of the round just completed to the aggregate representing the game; (ii) a complete rewrite of the whole game. The former is clearly more efficient. It is therefore useful to decompose aggregates into smaller data access units, according to the granularity of some important operations. In our example, we model each round of a game as a separate data unit.

In the next step of the methodology, aggregates and their smaller data access units are represented with NoAM, an abstract data model for NoSQL databases. This is a distinguishing feature of our approach: we use a data representation that refers to NoSQL databases but it is still independent of the actual systems. Indeed, the NoAM model defines abstractions of some major common features of NoSQL systems: (i) atomic operations on units of data access and distribution (corresponding to records/rows in extensible record stores, documents in document stores, and groups of key-value pairs in key-value stores), and (ii) data access operations on portions of such units (columns in extensible record stores, fields in document stores, and individual key-value pairs in key-value stores). Accordingly, the NoAM abstract data model has (i) *blocks*, which are units of data access and distribution, and (ii) *entries* of blocks, each of which associates a key with a (possibly complex) value. In this phase the aggregates and their smaller data access units identified in the previous step are mapped into blocks and entries of the NoAM abstract model. For example, a possible representation of the aggregates of Fig. 1 in the NoAM data model is shown in Fig. 4, where outer boxes denote blocks

representing aggregates, while inner boxes show entries. As mentioned before, the same data can be represented in different ways. To this end, we also propose design guidelines to select a suitable data representation, by taking into account the data access patterns of the application.

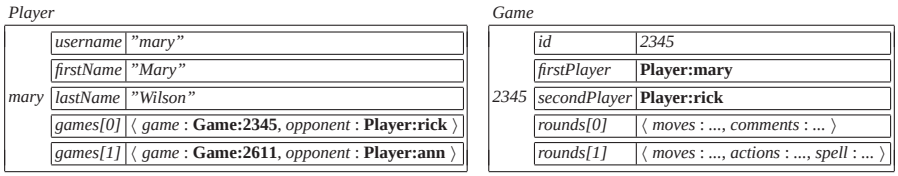

Fig. 4. A sample database in the abstract data model (abridged)

In the last step, the selected data representation in the NoAM abstract model is implemented using the specific data structures of a NoSQL system. 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 related key-value pairs. Figure 5 shows how the abstract database of Fig. 4 can be mapped to Oracle NoSQL.

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"**}
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*: ...}

Fig. 5. Implementation in Oracle NoSQL for the sample database of Fig. 4 (abridged)

3 The NoAM Abstract Data Model

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 of systems can be identified [9, 22]: key-value stores, extensible record stores, document stores, plus others that are beyond the scope of this paper.

NoAM (*NoSQL Abstract Data Model*) exploits the commonalities of such systems and introduces abstractions to balance their differences and variations. It 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*.

Figure 4 shows a sample NoAM database. In the figure, inner boxes show entries, while outer boxes denote blocks. Collections are shown as groups of blocks.

In NoAM, a *block* is a construct that models a data access and distribution unit, a concept available in all NoSQL systems. With reference to major NoSQL categories, a block corresponds to: (i) a record/row, in extensible record stores; (ii) a document, in document stores; or (iii) a group of related key-value pairs, in key-value stores. By "data access unit" we mean a *maximal* data unit that can be accessed and manipulated in an atomic, efficient, and scalable way. Indeed, most NoSQL systems do not provide atomic operations over multiple blocks (e.g., MongoDB [16] provides only atomic operations over individual documents) and queries that need to access multiple blocks, such as joins, can be quite inefficient. By "distribution unit" we mean that each unit is entirely stored in a node of the cluster, whereas different units are distributed among the various nodes.

In NoAM, an *entry* models the ability to access and manipulate just a component of a block. It corresponds to: (i) an attribute, in extensible record stores; (ii) a field, in document stores; or (iii) an individual key-value pair, in key-value stores. Note that entry values can be complex.

Finally, a NoAM *collection* groups data access units. For example, a table in extensible record stores or a document collection in document stores.

4 Conceptual Modeling and Aggregate Design

As discussed in Sect. 2, our methodology starts, as it is usual in database design, by building a conceptual representation of the data of interest. Following Domain-Driven Design (DDD [11]), which is a popular 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*. A *value object* is a persistent object which is mainly characterized by its value, without an own identifier.

For example, our application should manage players, games, and rounds (Fig. 1).

Then the methodology proceeds by identifying aggregates [11]. Each *aggregate* is a "chunk" of related data, with a complex value and a unique identifier, intended to represent a unit of data access and manipulation for an application. Aggregates are also important to support scalability and consistency, as they provide a natural unit for sharding and atomic manipulation of data in distributed environments [13, 11]. An important intuition in our approach is that each aggregate can be conveniently mapped to a NoAM block (Sect. 3), which is also a unit of data access and distribution. Aggregates and blocks are however distinct concepts, since they belong, respectively, to the application level and the database level.

Various approaches to aggregate design are possible. For example, in DDD [11], entities and value objects are then grouped into aggregates. Each *aggregate* has an entity as its root, and optionally it contains many value objects. Intuitively, an entity and a group of value objects define an aggregate having a complex structure and value.

Aggregate design is mainly driven by data access operations. In our running example, 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. Figure 1 also shows how application objects can be grouped in aggregates; there, a closed curve denotes the boundary of an aggregate.

Aggregate design is also driven by consistency needs. Specifically, aggregates should be designed as the units on which atomicity must be guaranteed [13] (with eventual consistency for update operations spanning multiple aggregates [20]). 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. A game (comprising, as an aggregate, its rounds) can check the above condition, while an individual round cannot. Therefore, a round cannot be an aggregate by itself.

Let us now illustrate the terminology we use to describe data at the aggregate level. An *application dataset* includes a number of *aggregate classes*, each having a distinct name. The extent of an *aggregate class* is a set of *aggregate objects* (or, simply, *aggregates*). Each aggregate has a *complex value* [1] and a unique *identifier*. In conclusion, our application has aggregate classes *Player* and *Game*.

5 Data Representation in NoAM and Aggregate Partitioning

In our approach, we use the NoAM data model as an intermediate model between application datasets of aggregates and NoSQL databases. Specifically, an application dataset can be represented by a NoAM database as follows. We represent each aggregate class by means of a distinct collection, and each aggregate object 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 application dataset of Fig. 1 can be represented by the NoAM database shown in Fig. 4. 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 representational choice propagates such qualities to operations on aggregates.

In general, an application dataset can be represented by a NoAM database in several ways. The various data representations for a dataset differ in the choice of the entries used to represent the complex value of each aggregate.

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. Each entry represents a distinct portion of the complex value v, characterized by a location in its structure (specified by the entry key) and a value (the entry value).

Aggregate partitioning can be driven by the following guidelines (which are a variant of guidelines proposed in [6] 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, the data representation for games shown in Fig. 4 is motivated by the following operation: when a player completes a round in a game she is playing, then the aggregate for the game should be updated. In order to update the underlying database, there would be two alternatives: (i) the addition of the round just completed to the aggregate representing the game; (ii) a complete rewrite of the whole game. The former is clearly more efficient. Therefore, each round is a candidate to be represented by an autonomous entry.

6 Implementation

In the last step, the selected data representation in NoAM is implemented using the specific data structures of a target datastore. For the sake of space, we discuss the implementation only with respect to a single system: Oracle NoSQL. We have also implementations for other systems [7, 8].

Oracle NoSQL [18] is a key-value store, in which a database is a schemaless collection of key-value pairs, with a key-value index. *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. On the other hand, each *value* is an uninterpreted binary string.

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).

A NoAM database D can be implemented in Oracle NoSQL 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. The value associated with this key is a representation of the entry value ev. The value can be either simple or a serialization of a complex value, e.g., in JSON.

For example, Fig. 5 shows the implementation of the data representation of Fig. 4.

An implementation can be considered *effective* if aggregates are indeed turned into units of data access and distribution. The effectiveness of this implementation is based on how we use major keys and minor keys to control distribution and consistency.

7 Discussion

The NoAM methodology for the design of NoSQL databases relies on an aggregate-oriented view of application data, an intermediate system-independent data model for NoSQL datastores, and an implementation activity that takes into account the features of specific systems. The overall approach aims at supporting the typical requirements of applications that can benefit from NoSQL technologies: scalability (selecting aggregates as units of distribution), consistency (assuming that aggregates are also units of atomic consistency), and performance (proposing different strategies for data representation according to data access operations).

We ran a number of experiments to compare the various data representation strategies in situations of different application workloads and database sizes, and measured the running time required by the workloads [7]. We also performed other experiments on a data representation that does not conform to the design guidelines proposed in this paper. The target system was Oracle NoSQL, a key-value store, deployed over Amazon AWS on a cluster of four EC2 servers.

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

Currently, we are developing a tool that provides a common programming interface towards different NoSQL systems, to access them in a unified way, in the spirit of SOS [2]. The tool uses an internal representation based on NoAM, and it also supports the design approach presented in this paper. We also plan to extend our methodology to include data duplication to support query-based workloads.

References

1. S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.
2. P. Atzeni, F. Bugiotti, and L. Rossi. Uniforma access to NoSQL systems. *Inf. Syst.*, 43, 117–133, 2014.
3. P. Atzeni, C. S. Jensen, G. Orsi, S. Ram, L. Tanca, and R. Torlone. The relational model is dead, SQL is dead, and I don't feel so good myself. *SIGMOD Record*, 42(2):64–68, 2013.
4. A. Badia and D. Lemire. A call to arms: revisiting database design. *SIGMOD Record*, 40(3):61–69, 2011.
5. J. Baker et al. Megastore: Providing scalable, highly available storage for interactive services. In *CIDR 2011*, pages 223–234, 2011.
6. C. Batini, S. Ceri, and S. B. Navathe. *Conceptual Database Design: An Entity-Relationship Approach*. Benjamin/Cummings, 1992.
7. F. Bugiotti, L. Cabibbo, P. Atzeni, and R. Torlone. Database design for NoSQL systems. In *ER 2014*, pages 223–231, 2014.
8. F. Bugiotti, L. Cabibbo, R. Torlone, and P. Atzeni. Database design for NoSQL systems. Technical Report 210, Università Roma Tre, 2014. Available from *http://www.inf.uniroma3.it/?page_id=476*.
9. R. Cattell. Scalable SQL and NoSQL data stores. *SIGMOD Record*, 39(4):12–27, 2010.
10. F. Chang et al. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26(2), 2008.
11. E. Evans. *Domain-Driven Design*. Addison-Wesley, 2003.
12. M. Hamrah. Data modeling at scale. 2011.
13. P. Helland. Life beyond distributed transactions: an apostate's opinion. In *CIDR 2007*, pages 132–141, 2007.
14. I. Katsov. NoSQL data modeling techniques. 2012. Highly Scalable Blog.
15. C. Mohan. History repeats itself: sensible and NonsenSQL aspects of the NoSQL hoopla. In *EDBT*, pages 11–16, 2013.
16. MongoDB Inc. MongoDB. *http://www.mongodb.org*. Accessed 2014.
17. T. Olier. Database design using key-value tables. 2006.
18. Oracle. Oracle NoSQL Database. *http://www.oracle.com/technetwork/products/nosqldb*. Accessed 2014.
19. J. Patel. Cassandra data modeling best practices. 2012.
20. D. Pritchett. BASE: An ACID alternative. *ACM Queue*, 6(3):48–55, 2008.
21. A. Rathore. HBase: On designing schemas for column-oriented data-stores. 2009.
22. P. J. Sadalage and M. J. Fowler. *NoSQL Distilled*. Addison-Wesley, 2012.
23. M. Stonebraker. Stonebraker on NoSQL and enterprises. *Comm. ACM*, 54(8):10–11, 2011.