Database Design for NoSQL Systems

Francesca Bugiotti, Luca Cabibbo, Paolo Atzeni, Riccardo Torlone

Database Design for NoSQL Systems

Francesca Bugiotti[1]*, Luca Cabibbo[2], Paolo Atzeni[2], and Riccardo Torlone[2]

[1]Inria & Université Paris-Sud and [2]Università Roma Tre

Abstract. We propose a database design methodology for NoSQL systems. The approach is based on NoAM (NoSQL Abstract Model), a novel abstract data model for NoSQL databases, which exploits the commonalities of various NoSQL systems and 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. 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 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) scalability and performance are important quality requirements, and (iv) a certain level of consistency is also desirable [7, 20].

NoSQL technology is characterized by a high heterogeneity [7, 21], which is problematic to application developers. Currently, database design for NoSQL systems is usually based on best practices and guidelines [12], which are specifically related to the selected system [19, 10, 17], with no systematic methodology. Several authors have observed that the development of high-level methodologies and tools supporting NoSQL database design are needed [2, 3, 13].

In this paper we aim at filling this gap, by presenting a design methodology for NoSQL databases that has initial activities that are independent of the specific target system. The approach is based on *NoAM* (*NoSQL Abstract Model*), a novel abstract data model for NoSQL databases, which exploits the observation that the various NoSQL systems share similar modeling features. Given the application data and the desired data access patterns, the methodology we propose uses NoAM to specify an intermediate, system-independent data representation. The implementation in target NoSQL systems is then a final step, with a translation that takes into account their peculiarities.

Specifically, our methodology has the goal of designing a "good" representation of these application data in a target NoSQL database, and is intended to support *scalability*, *performance*, and *consistency*, as needed by next-generation web applications. In general, different alternatives on the organization of data in a NoSQL database are possible, but they are not equivalent in supporting performance, scalability, and consistency. A "wrong" database representation can lead to the inability to guarantee atomicity of important operations and to performance that are worse by an order of magnitude.

The design methodology is based on the following main activities:

*Part of this work was performed while this author was with Università Roma Tre.

– *conceptual data modeling*, to identify the various entities and relationships thereof needed in an application;
– *aggregate design*, to group related entities into aggregates [9, 11];
– *aggregate partitioning*, where aggregates are partitioned into smaller data elements;
– *high-level NoSQL database design*, where aggregates are mapped to the NoAM intermediate data model, according to the identified partitions;
– *implementation*, to map the intermediate data representation to the specific modeling elements of a target datastore; only this activity depends on the target system.

The remainder of this paper presents our methodology for NoSQL database design. As a running example, we consider an application for an on-line social game. This is a typical scenario in which the use of a NoSQL database is suitable. For space reasons, many details have been omitted; they can be found in the full version of the paper [6].

2 The NoAM Abstract Data Model

In this section we present the NoAM abstract data model for NoSQL databases. Preliminarily, we briefly sum up the data models used in NoSQL databases.

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 according to their modeling features [7, 20]: key-value stores, extensible record stores, document stores, plus others that are beyond the scope of this paper.

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 (e.g., sharing part of the key). The key (or part of it, thereof) controls data distribution.

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 (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.

In a *document store*, a database is a set of documents, each having a complex structure and value. Documents are organized in collections. Operations usually access individual documents, which are units of data distribution and atomic data manipulation.

NoAM (*NoSQL Abstract Data Model*) is a novel data model for NoSQL databases that exploits the commonalities of the data modeling elements available in the various NoSQL systems and introduces abstractions to balance their differences and variations.

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*.

Figure 1 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, which is a data modeling element available in all NoSQL systems. By “data access

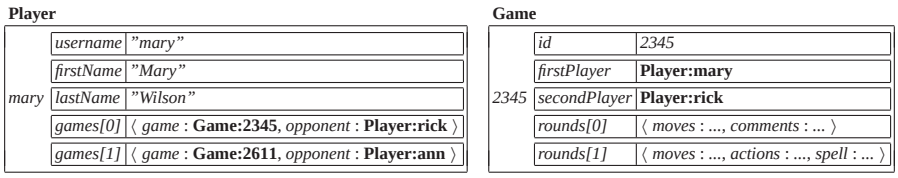


Fig. 1. A sample database in the abstract data model (abridged)

unit" we mean that the NoSQL 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, 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.

Specifically, a block represents a *maximal* data unit for which atomic, efficient, and scalable access operations are provided. Indeed, in the various systems, the access to multiple blocks can be quite inefficient. For example, NoSQL systems do not provide an efficient "join" operation. Moreover, most NoSQL systems do not provide atomic operations over multiple blocks. For example, MongoDB [14] provides only atomic operations over individual documents.

In NoAM, an *entry* models the ability to access and manipulate just a component of a data access unit (i.e., of a block). An entry is a smaller data unit that 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* models a collection of data access units. For example, a table in extensible record stores or a document collection in document stores.

In summary, NoAM describes in a uniform way the features of many NoSQL systems. We will use it for an intermediate representation in the design process.

3 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. See, for example, [5]. Following Domain-Driven Design (DDD [9]), 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 various types of objects, including players, games, and rounds. A few representative objects are shown in Fig. 2. (Consider, for now, only boxes and arrows, which denote objects and links between them.)

The methodology proceeds by identifying aggregates [9]. Intuitively, 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 [11, 9]. An important intuition in our approach is that each aggregate can be conveniently mapped to

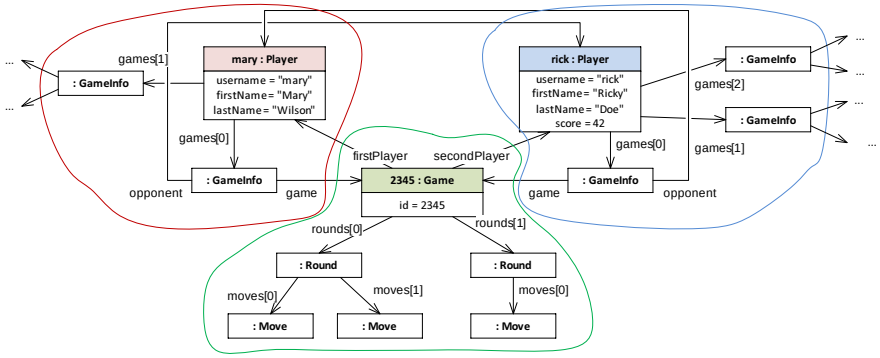


Fig. 2. Sample application objects

a NoAM block (Sect. 2), 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 [9], 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.

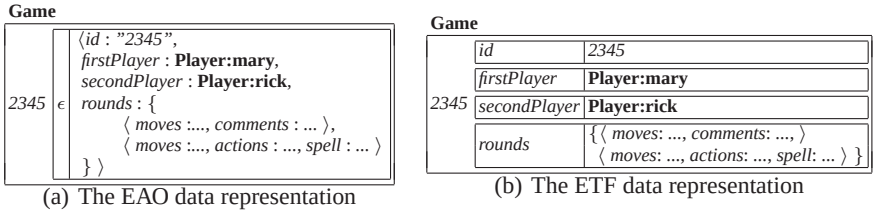
Aggregate design is mainly driven by data access operations. In our running example, when a player connects to the application, all data on the player should be retrieved, including an overview of the games she is currently playing. Then, the player can select to continue a game, and data on the selected 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 2 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 [11] (with eventual consistency for update operations spanning multiple aggregates [18]). 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**.

4 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



(a) The EAO data representation

(b) The ETF data representation

Fig. 3. Data representations (abridged)

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. 2 can be represented by the NoAM database shown in Fig. 1. 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.

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. 2 according to the EAO strategy is shown in Fig. 3(a). (For the sake of space, we show only the data representation for the game aggregate object.)

Another 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. See Fig. 3(b).

The data representation strategies described above can be suited in some cases, but they are often too rigid and limiting. The main limitation of such general representations is that they refer only to the structure of aggregates, and do not take into account the required data access operations. Therefore, they do not usually support the performance of these operations. This motivates the introduction of aggregate partitioning.

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). We have already applied this intuition in the ETF data representation (shown in Fig. 3(b)), which uses field names as entry keys and field values as entry values.

Aggregate partitioning can be driven by the following guidelines (which are a variant of guidelines proposed in [5] 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.

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

(a) EAO in Oracle NoSQL

key (/major/key/-/minor/key)	*value*
/Game/2345/-/id	*2345*
/Game/2345/-/firstPlayer	*Player:mary*
/Game/2345/-/secondPlayer	*Player:rick*
/Game/2345/-/rounds	*[{ ... }, { ... }]*

(b) ETF in Oracle NoSQL

Fig. 4. Implementation in Oracle NoSQL (abridged)

key (/major/key/-/minor/key)	*value*
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. 1 (abridged)

- 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. 1 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.

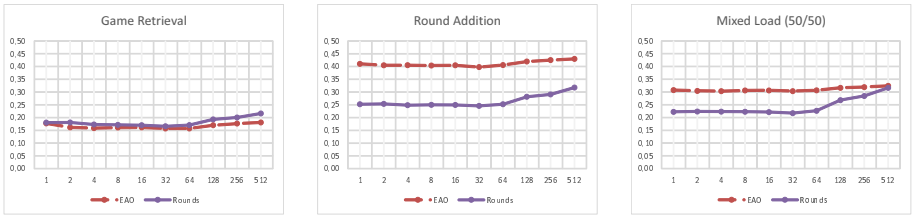
5 Implementation

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 [6].

Oracle NoSQL [16] 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.

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. 4(a) and 4(b) show the implementation of the EAO and ETF data representations, respectively, in Oracle NoSQL. Moreover, Fig. 5 shows the implementation of the data representation of Fig. 1.



(a) Game Retrieval (b) Round Addition (c) Mixed Load

Fig. 6. Experimental results

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 the fact that 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).

6 Experiments

We now discuss a case study of NoSQL database design, with reference to our running example. For the sake of simplicity, we focus only on the representation 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. To manage games, the candidate data representations are: (i) using a single entry for each game (as shown in Fig. 3(a), 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 (a variant of the representation shown in Fig. 1, called ROUNDS).

We ran a number of experiments to compare the above data representations in situations of different application workloads and database sizes, and measured the running time required by the workloads. The target system was Oracle NoSQL, a key-value store, deployed over Amazon AWS on a cluster of four EC2 servers. (This work was supported by AWS in Education Grant award.)

The results are shown in Fig. 6. Database sizes are in gigabytes, timings are in milliseconds, and points denote the average running time of a single operation. The experiments show that the retrieval of a game (Fig. 6(a)) is always favored by the EAO data representation, for any database size. They also show that the addition of a round to an existing game (Fig. 6(b)) is always favored by the ROUNDS data representation. Finally, the experiments over the mixed workload (Fig. 6(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, we divided the rounds of a game into independent key-value pairs, rather than keeping them together in a same block. In this case, the performance of the various operations worsened by an order of magnitude. Moreover, it was 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.

7 Related Work

Several authors have observed that the development of methodologies and tools supporting NoSQL database design is demanding [2, 3, 13]. However, this topic has been explored so far only in some on-line papers, published in blogs of practitioners, in terms of best practices and guidelines for modeling NoSQL databases (e.g., [12, 15]), and usually with reference to specific systems (e.g., [19, 10, 17]). To the best of our knowledge, this is the first proposal of a system-independent approach to the design of NoSQL databases, which tackles the problem from a general perspective.

Domain-Driven Design [9] is a widely followed object-oriented approach that includes a notion of aggregate. Also [11] advocates the use of aggregates (there called entities) as units of distribution and consistency. We also propose, for efficiency purposes, to partition aggregates into smaller units of data access and manipulation.

In [4] the authors propose entity groups, a set of entities that, similarly to our aggregates, can be manipulated in an atomic way. They also describe a specific mapping of entity groups to Bigtable [8]. 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.

References

1. S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.
2. 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.
3. A. Badia and D. Lemire. A call to arms: revisiting database design. *SIGMOD Record*, 40(3):61–69, 2011.
4. J. Baker et al. Megastore: Providing scalable, highly available storage for interactive services. In *CIDR 2011*, pages 223–234, 2011.
5. C. Batini, S. Ceri, and S. B. Navathe. *Conceptual Database Design: An Entity-Relationship Approach*. Benjamin/Cummings, 1992.
6. 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.dia.uniroma3.it/Plone/ricerca/technical-reports/2014*.
7. R. Cattell. Scalable SQL and NoSQL data stores. *SIGMOD Record*, 39(4):12–27, 2010.
8. F. Chang et al. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26(2), 2008.
9. E. Evans. *Domain-Driven Design*. Addison-Wesley, 2003.
10. M. Hamrah. Data modeling at scale. 2011.
11. P. Helland. Life beyond distributed transactions: an apostate's opinion. In *CIDR 2007*, pages 132–141, 2007.
12. I. Katsov. NoSQL data modeling techniques. 2012. Highly Scalable Blog.
13. C. Mohan. History repeats itself: sensible and NonsenSQL aspects of the NoSQL hoopla. In *EDBT*, pages 11–16, 2013.
14. MongoDB Inc. MongoDB. *http://www.mongodb.org*. Accessed 2014.
15. T. Olier. Database design using key-value tables. 2006.
16. Oracle. Oracle NoSQL Database. *http://www.oracle.com/technetwork/products/nosqldb*. Accessed 2014.
17. J. Patel. Cassandra data modeling best practices. 2012.
18. D. Pritchett. BASE: An ACID alternative. *ACM Queue*, 6(3):48–55, 2008.
19. A. Rathore. HBase: On designing schemas for column-oriented data-stores. 2009.
20. P. J. Sadalage and M. J. Fowler. *NoSQL Distilled*. Addison-Wesley, 2012.
21. M. Stonebraker. Stonebraker on NoSQL and enterprises. *Comm. ACM*, 54(8):10–11, 2011.