Database Design for NoSQL Systems

Francesca Bugiotti, Luca Cabibbo, Paolo Atzeni, Riccardo Torlone

ROMA TRE
UNIVERSITÀ DEGLI STUDI

DIPARTIMENTO DI INFORMATICA E AUTOMAZIONE
Via della Vasca Navale, 79
00146 Roma, Italy

Database Design for NoSQL Systems

FRANCESCA BUGIOTTI[1], LUCA CABIBBO[2], PAOLO ATZENI[2], RICCARDO TORLONE[2]

RT-DIA-210-2014 **June 2014**

(1) Inria & Université Paris-Sud,
Batiment 650 (PCRI)
91405 Orsay Cedex, France

(2) Università Roma Tre,
Via della Vasca Navale, 79
00146 Roma, Italy

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

ABSTRACT

The popularity of NoSQL database systems is rapidly increasing, especially to support next-generation web applications. However, given the high heterogeneity existing in this world, where more than fifty systems are available, database design is usually based on best practices and guidelines which are strictly related to the selected system.

We propose a database design methodology for NoSQL systems with 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 commonalities of various NoSQL systems and is used to specify a system-independent representation of the application data. We show how 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) scalability and performance are important quality requirements, and a certain level of consistency is also desirable [8, 21].

More than fifty NoSQL systems exist [23], each with different characteristics. They can be classified into a few main categories [8], including key-value stores, document stores, and extensible record stores. In any case, heterogeneity is highly problematic to application developers [23], even within each category.

Currently, database design for NoSQL systems is usually based on best practices and guidelines [13], which are specifically related to the selected system [20, 11, 18], with no systematic methodology. Several authors have observed that the development of high-level methodologies and tools supporting NoSQL database design are needed [4, 5, 14]. Indeed, different alternatives on the organization of data in NoSQL databases exist, with significant consequences on major quality requirements, including scalability, performance, and consistency [13].

In this paper, we present 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. NoAM exploits the observations that the various NoSQL systems share similar modeling features. An important insight is that each NoSQL system offers efficient, atomic, and scalable access operations on "data access units" at a certain granularity. 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.

The design methodology is intended to support *scalability*, *performance*, and *consistency*, as needed by next-generation web applications. To this end, a major observation is that it is useful to arrange application data in *aggregates* [10, 12], that is, groups of related data, with a complex value, representing units of data access and atomic manipulation. Aggregates can then be managed in the various NoSQL systems, as their "data access units" are compatible with the features of aggregates. This way, the efficient, atomic, and scalable data access operations can support performance, consistency, and scalability of simple read-write operations over the aggregates of the application.

In accordance with the above observations, the NoAM approach is based on the following main activities:

- *conceptual data modeling*, to identify the various entities and relationships thereof needed in an application;
- *aggregate design*, to group related entities into aggregates;
- *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;

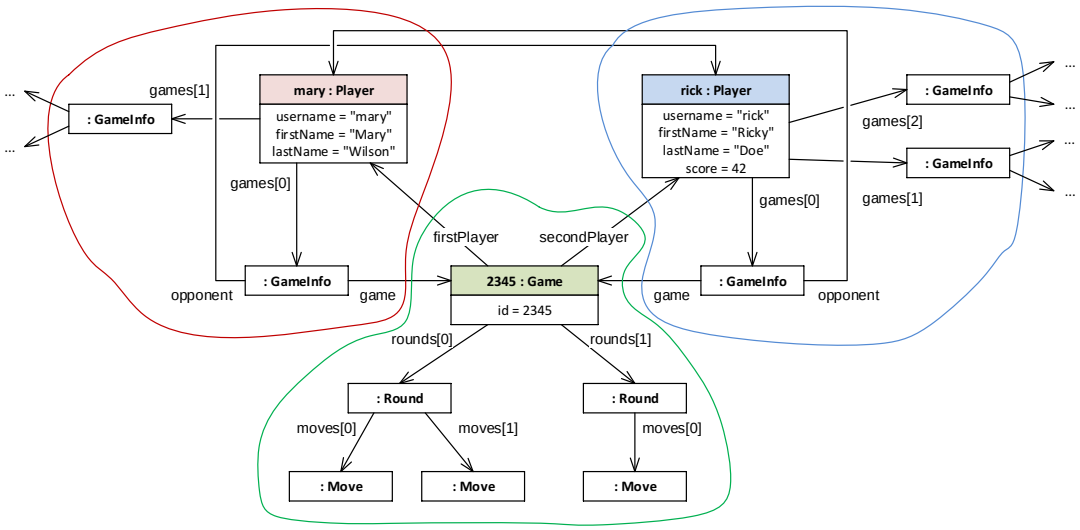


Figure 1: Sample application objects

- *implementation*, to map the intermediate data representation to the specific modeling elements of a target datastore.

We point out that only the implementation depends on the target datastore.

The paper is organized as follows: Section 2 provides an overview of our contribution. Section 3 presents the various data models that we use in our approach. Section 4 discusses conceptual modeling and aggregate design. Section 5 introduces data representation strategies for aggregates in our abstract data model. Section 6 presents some guidelines to partition aggregates and discusses data representations further. Section 7 describes how data representations can be implemented in target NoSQL systems. Section 8 illustrates a case study of NoSQL database design. Section 9 discusses related work. Finally, Section 10 draws some conclusion.

2 Overview

Let us consider, as a running 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 methodology starts, as it is usual in database design, by building a conceptual representation of the data of interest. For example, our application should manage various types of objects, including players, games, and rounds. A few representative objects are shown in Figure 1. (There, boxes and arrows denote objects and relationships between them, respectively; please ignore, for now, colors and closed curves.)

Our methodology has the goal of designing a "good" representation of these application data in a target NoSQL database. In general, different alternatives are possible, but they are not equivalent in supporting performance, scalability, and consistency. A "wrong" database representation can lead to performance that are worse by an order of magnitude, and to the inability to guarantee atomicity of important operations.

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 distributed environments [12, 10]. In our example, aggregates

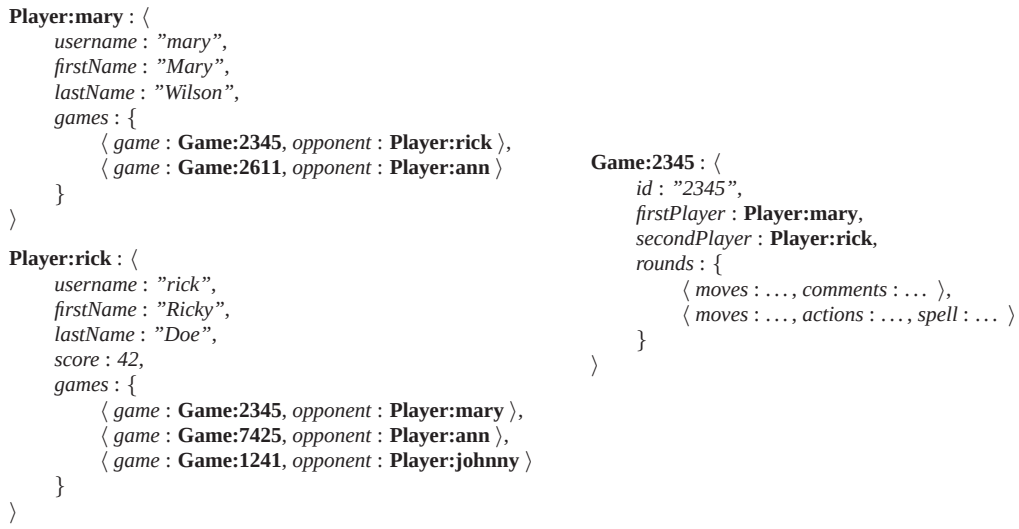


Figure 2: Sample aggregates (as complex values)

are players and games, as shown by closed curves in Figure 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 Figure 2.

The following activity consists in partitioning aggregates, if needed, into smaller elements to better support the performance of certain operations. Consider 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. This fact suggests that it is useful to decompose aggregates into smaller data access units, according to the granularity of some important operations. In our example, we represent each round of a game as a separate data unit.

The next steps of the methodology are based on 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 the common data modeling features provided by the various NoSQL systems. These abstractions are based on the observation that NoSQL systems offer: (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. Then, a block is a collection of (ii) *entries*, each of which associates a key with a (possibly complex) value. Blocks have their own keys and are grouped in collections.

The approach continues by representing aggregates and their smaller data access units into blocks and entries of the NoAM abstract model. This mapping is justified by the fact that aggregates in the application should be managed as data access and distribution units — and that blocks represent data modeling elements provided by NoSQL databases that are, indeed, data access and distribution units. For example, Figure 3 shows a possible representation of the aggregates of Figure 1 in terms of the NoAM data model. There, outer boxes denote blocks representing aggregates, while inner boxes show entries. As mentioned before, the same data

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* : ... ⟩

Figure 3: A sample database in the abstract model

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*: ...}

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

can be represented in different ways. Compare, for example, Figure 3 with Figures 6 and 7 later. We also propose design guidelines to select a suitable data representation, by taking into account the data access patterns required by the application.

In the last step, the selected data representation in the NoAM abstract model is implemented using the specific data structures of a target datastore. For example, if the target system is a key-value store, then each entry is mapped to a distinct key-value pair, while blocks correspond to groups of key-value pairs, sharing part of the key. Figure 4 shows how the NoAM database of Figure 3 can be mapped to Oracle NoSQL. An implementation can be considered "effective" if aggregates are indeed turned into units of data access and distribution. The implementation shown in Figure 3 is effective in this sense, by the use we make of Oracle NoSQL keys, which control distribution and atomicity of operations. If the target system is an extensible-record store, such as DynamoDB [2], then each block is mapped to an item and each of its entries is mapped to an attribute of the item, as shown in Figure 5.

3 Data Models

In our approach, data are represented using various data models, at four different layers: (i) at the application level, data are organized as *application objects*; (ii) application objects are grouped into *aggregates*, which are complex-value objects; (iii) aggregates are represented in a system-independent way, according to the NoAM *abstract data model* for NoSQL databases;

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*: ... }	{ *moves*: ..., *actions*: ..., *spell*: ... }	

Figure 5: Implementation in DynamoDB for the sample database of Figure 3 (abridged)

(iv) finally, the abstract database representation is mapped to the specific data modeling elements of a selected target NoSQL system. This section describes these data models, in a bottom-up order.

3.1 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 [8, 21]: 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 (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.

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

3.2 The NoAM Abstract Data Model

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.

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 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, this element is: (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.

In NoAM, we have a construct that models a data access and distribution unit. It is called 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 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 [15] provides only atomic operations over individual documents, whereas Bigtable does not support transactions across rows [9].

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) a column, in extensible record stores; (ii) a field, in document stores; or (iii) an individual key-value pair, in key-value stores. In NoAM, 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*.

Figure 3 shows a sample NoAM database. In the figure, inner boxes show entries, while outer boxes denote blocks. Collections are shown as groups of blocks.

Note that entry values can be complex values. The practice of storing a complex value as an individual data element is common to various NoSQL systems; examples are Protocol Buffers [22] and Avro schemas [17].

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

3.3 Aggregate Data Model

In the context of next-generation web applications, it is useful to consider application data organized in *aggregates* [10]. Intuitively, each aggregate is a "chunk" of related data, with a complex value and a unique identifier, which is intended to represent a unit of data access and manipulation for an application. Moreover, to support scalability, aggregates should also govern data distribution, as suggested by best practices in the design of scalable applications [12, 10]. An important intuition in our approach is that each aggregate can be conveniently mapped to a NoAM block, which is also a unit of data access and distribution (Section 3.2). Aggregates and blocks are however distinct concepts, since they belong, respectively, to the application and the database levels.

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* and an *identifier* (which is unique within the aggregate class the object belongs to). Complex values [1] are built using basic values (e.g., numbers and strings), references to aggregates (of the form $C : id$, where C is an aggregate class and id is the identifier of an aggregate), records, and collections. We assume that the complex value of each aggregate is,

at the top level, a record. We also assume that, for each aggregate class, there exists a top-level field to hold a basic value as the identifier for the aggregates in that class. References are used to represent (unidirectional) relationships between aggregates.

3.4 Application Data Model

At the application level, we assume that data are modeled in a conceptual way [7, 10], using an object-based data model, in terms of entities, value objects, and relationships. An *entity* is a persistent object that has independent existence and is distinguished by a unique *identifier*. A *value object* is a persistent object which is mainly characterized by its value, without an own identifier.

4 Conceptual Modeling and Aggregate Design

In our approach, conceptual modeling is performed in a standard way. See, for example, [7]. Following Domain-Driven Design (DDD [10]), 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 (see Section 3.4). Thus, the remainder of this section focuses on aggregate design.

The design of aggregates has the goal of identifying the classes of aggregates for an application, and various approaches are possible. For example, this activity is described by DDD [10]. After a 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. In our approach, this activity is driven by the data access patterns of the application operations, as well as by scalability and consistency needs [10]. Specifically, we consider individual aggregates as the units on which atomicity must be guaranteed [12] (with eventual consistency for update operations spanning multiple aggregates [19]). In general, it is indeed the case that most real applications require only operations that access individual aggregates [8, 9]. Each aggregate should be large enough so as to include all the data required by a relevant data access operation. (Please 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 [24]. On the other hand, aggregates should be as small as possible; small aggregates reduce concurrency collisions and support performance and scalability requirements [24].

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. Figure 1 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. Figure 1 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 dataset for our application consists of aggregate classes **Player** and **Game**, as shown in Figures 1 and 2.

5 Data Representation in NoAM

In our approach, we use the NoAM data model (Section 3.2) as an intermediate model between application datasets of aggregates (Section 3.3) and NoSQL databases (Section 3.1). This issue is discussed in this section and in the following Section 6.

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 Figures 1 and 2 can be represented by the NoAM database shown in Figure 3. 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 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. This section is devoted to the discussion of basic data representation strategies, which we illustrate with respect to the example described in Figure 2. Additional and more flexible data representations will be discussed in Section 6.

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 Figure 2 according to the EAO strategy is shown in Figure 6.

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. Figure 7 shows the data representation of the aggregates of Figure 2 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).

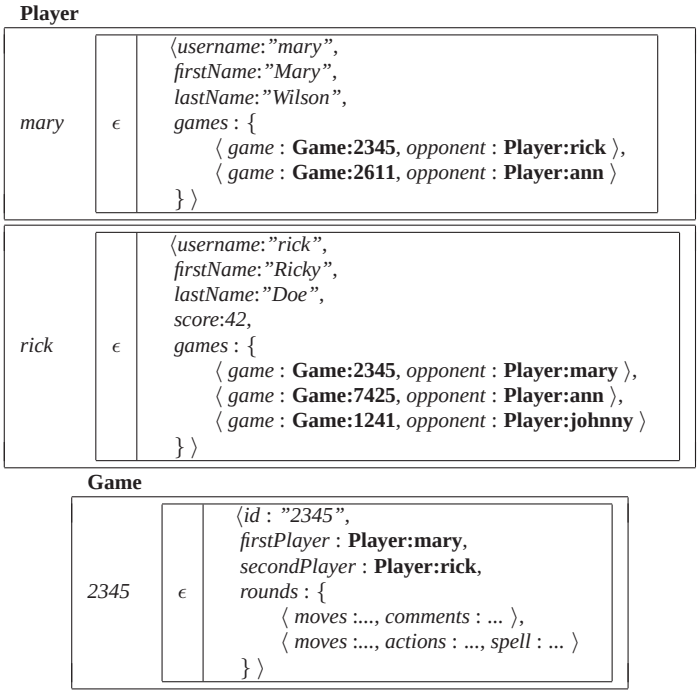


Figure 6: The EAO data representation

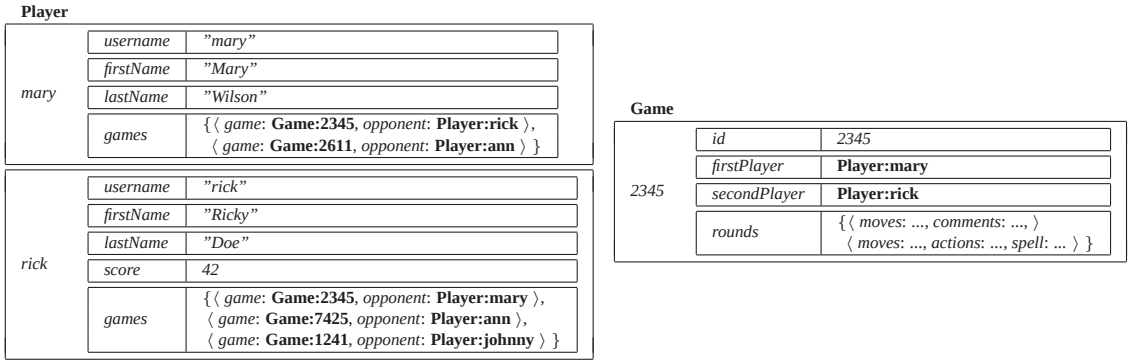


Figure 7: The ETF data representation

6 Aggregate Partitioning

The general data representation strategies introduced in the previous section 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 Figure 3. The main limitation of such general data representations 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, which we discuss in this section.

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 v' is a value (possibly complex as well) occurring in v, then the access path ap for v' in v represents the sequence of "steps" that should be taken to reach the component value v' 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 Figure 2. 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 Figure 2. A possible entry for v_{mary} is the pair $\langle$*games[0].opponent*, **Player:rick**$\rangle$. We have already applied the above intuition in Section 5. For example, the ETF data representation (shown in Figure 7) 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 [7] 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 should be implemented in the database just as an addition of a round to a game, rather than a complete rewrite of the whole game. 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 Figure 3. 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 Figure 8.

7 Implementation

In this section, we show how a NoAM data representation can be implemented in a target NoSQL database. Given that NoAM generalizes the features of the various NoSQL systems,

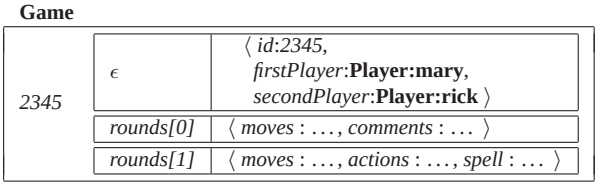


Figure 8: An alternative data representation for games (ROUNDS)

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 categories. Moreover, with reference to the same aggregate objects of Figures 1 and 2 we will sometimes show only the data for one aggregate. Similar representations can be obtained for the other aggregates of the running example.

7.1 Key-Value Store: Oracle NoSQL

Oracle NoSQL [17] 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. Each element of a key is called a *component* of the key. On the other hand, each *value* is an uninterpreted binary string.

A data representation D for a dataset 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 (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, Figures 9(a) and 9(b) show the implementation of the EAO and ETF data representations, respectively, in Oracle NoSQL. Moreover, Figure 4 shows the implementation of the data representation of Figure 3.

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 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 op-

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

(a) EAO in Oracle NoSQL

key (*/major/key/-/minor/key*)	*value*
/Player/mary/-/username	*mary*
/Player/mary/-/firstName	*Mary*
/Player/mary/-/lastName	*Wilson*
/Player/mary/-/games	*[* { ... }, { ... } *]*
/Player/rick/-/username	*rick*
/Player/rick/-/firstName	*Ricky*
/Player/rick/-/lastName	*Doe*
/Player/rick/-/score	*42*
/Player/rick/-/games	*[* { ... }, { ... }, { ... } *]*
/Game/2345/-/id	*2345*
/Game/2345/-/firstPlayer	*Player:mary*
/Game/2345/-/secondPlayer	*Player:rick*
/Game/2345/-/rounds	*[* { ... }, { ... } *]*

(b) ETF in Oracle NoSQL

Figure 9: Implementation in Oracle NoSQL

erations 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 including in each update operation the access to a distinguished entry of the block for an aggregate object; in particular, this could be the entry that includes the identifier of the aggregate, or a specific “version” field.

7.2 Extensible Record Store: DynamoDB

Amazon DynamoDB [2] is a NoSQL database service provided on the cloud by Amazon Web Services (AWS). DynamoDB is an extensible record store. A database is organized in tables. A *table* is a set of items. Each *item* contains one or more *attributes*, each with a *name* and a *value* (or a set of values). Each table designates an attribute as *primary key*. Items in a same table are not required to have the same set of attributes — apart from the primary key, which is the only mandatory attribute of a table. Thus, DynamoDB databases are mostly schemaless.

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

The implementation of a NoAM database in DynamoDB 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.

We now show how to implement a NoAM data representation D in DynamoDB. 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 will use all the entries of a block b to create a new item in a table for b. Specifically, we proceed as follows: (i) the collection name C is used as a 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, Figure 5 shows the implementation of the NoAM database of Figure 3.

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

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"**}] }*

Figure 10: 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"** }* }

Figure 11: Alternative implementation in MongoDB

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.

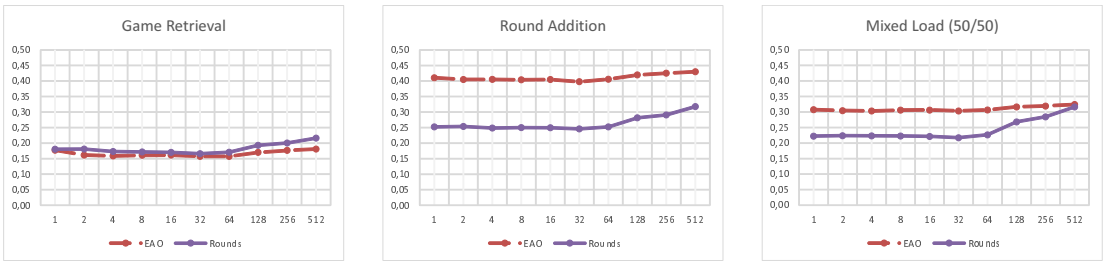
7.3 Document Store: MongoDB

MongoDB [15] is an open-source, document-oriented data store. In MongoDB, a database is made of *collections* of documents. Each *document* is a structured document, that is, a complex value, a set of field-value pairs, which can comprise simple values, lists, and even nested documents. A *main document* is a top-level document with a unique identifier, represented by a special field $\_id$. Documents are schemaless, as each document can have its own attributes, defined at runtime. Specifically, MongoDB documents are based on BSON (Binary JSON), a variant of the popular JSON format.

According to our approach, a natural implementation for a NoAM database in MongoDB 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 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*$\}$.

Specifically, 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 documento 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 Figure 10.

The retrieval of a block, given its collection C and key id, can be implemented by per-



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

Figure 12: Experimental results

forming 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 Figure 3 leads to the result shown in Figure 11.

8 Experiments

We will now discuss a case study of NoSQL database design, with reference to the running example introduced in Section 2. For the sake of simplicity, we just focus on the representation and management of aggregate objects for 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 Figure 6, 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 Figure 8, 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 8KB 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.

The results are shown in Figure 12. 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 (Figure 12(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 (Figure 12(b)) is favored by the ROUNDS data representation. Finally, the experiments over the mixed workload (Figure 12(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.

9 Related Work

To the best of our knowledge, this is the first proposal of a general, system-independent approach to the design of NoSQL databases. Indeed, although several authors have observed that the development of methodologies and tools supporting NoSQL database design is demanding [4, 5, 14], this topic has been not explored yet from a general point of view. The only examples are some on-line papers, usually published in blogs of practitioners, that discuss best practices and guidelines for modeling NoSQL databases, such as [13, 16]. Some papers are suited only for specific systems, e.g., [20, 11, 18]. None of them tackles the problem from a general perspective, as we do in this paper.

Our approach takes inspiration from Domain Driven Design [10], a widely followed object-oriented approach that includes a notion of aggregate. Also [12] 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 [6] 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 [9]. 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.

In [4] it has been observed that the availability of a high-level representation of the data remains a fundamental tool for developers and users, since it makes understanding, managing, accessing, and integrating information sources much easier, independently of the technologies used. The present paper responds to these needs.

10 Conclusion

In this paper, we have proposed a comprehensive methodology for the design of NoSQL databases. The approach 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.

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

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 [3]. The tool uses an internal representation based on NoAM, and it also supports the design approach presented in this paper. We believe that database designers can greatly benefit from such a tool, as we envision the use of the framework to tune the choice of the data representation in a flexible way, as well as to select the most suitable target NoSQL system.

References

[1] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[2] Amazon Web Services. DynamoDB. *http://aws.amazon.com/dynamodb*. Accessed 2014.

[3] Paolo Atzeni, Francesca Bugiotti, and Luca Rossi. Uniform access to non-relational database systems: The SOS platform. In *International Conference on Advanced Information Systems Engineering, CAiSE 2012*, pages 160–174, 2012.

[4] Paolo Atzeni, Christian S. Jensen, Giorgio Orsi, Sudha Ram, Letizia Tanca, and Riccardo Torlone. The relational model is dead, SQL is dead, and I don't feel so good myself. *SIGMOD Record*, 42(2):64–68, 2013.

[5] Antonio Badia and Daniel Lemire. A call to arms: revisiting database design. *SIGMOD Record*, 40(3):61–69, 2011.

[6] Jason Baker et al. Megastore: Providing scalable, highly available storage for interactive services. In *CIDR 2011*, pages 223–234, 2011.

[7] Carlo Batini, Stefano Ceri, and Shamkant B. Navathe. *Conceptual Database Design: An Entity-Relationship Approach*. Benjamin/Cummings, 1992.

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

[9] Fay Chang et al. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26(2), 2008.

[10] Eric Evans. *Domain-Driven Design*. Addison-Wesley, 2003.

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

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

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

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

[15] MongoDB Inc. MongoDB. *http://www.mongodb.org*. Accessed 2014.

[16] Tal Olier. Database design using key-value tables. *http://www.devshed.com/c/a/MySQL/Database-Design-Using-KeyValue-Tables/*, 2006.

[17] Oracle. Oracle NoSQL Database. *http://www.oracle.com/technetwork/products/nosqldb*. Accessed 2014.

[18] Jay Patel. Cassandra data modeling best practices. *http://www.ebaytechblog.com/-2012/07/16/cassandra-data-modeling-best-practices-part-1/*, 2012.

[19] Dan Pritchett. BASE: An ACID alternative. *ACM Queue*, 6(3):48–55, 2008.

[20] Amit Rathore. HBase: On designing schemas for column-oriented data-stores. *http://s-expressions.com/2009/03/08/hbase-on-designing-schemas-for-column-oriented-data-stores/*, 2009.

[21] Pramodkumar J. Sadalage and Martin J. Fowler. *NoSQL Distilled*. Addison-Wesley, 2012.

[22] Jeff Shute et al. F1: A distributed SQL database that scales. *PVLDB*, 6(11):1068–1079, 2013.

[23] Michael Stonebraker. Stonebraker on NoSQL and enterprises. *Comm. ACM*, 54(8):10–11, 2011.

[24] Vaughn Vernon. *Implementing Domain-Driven Design*. Addison-Wesley, 2013.