A Comparison of Data Models and APIs of NoSQL Datastores

Francesca Bugiotti and Luca Cabibbo

Dipartimento di Ingegneria
Università Roma Tre

Abstract. NoSQL datastore systems are a new generation of non-relational databases. More than fifty NoSQL systems have been already implemented, each with different characteristics — especially, with different data models and different APIs to access the data. In this paper we describe and compare the data models and operations offered by a number of representative NoSQL datastores, which we have directly used while developing the SOS (Save Our Systems) and ONDM (Object-NoSQL Datastore Mapper) frameworks. We discuss how these NoSQL systems can be used to manage a database consisting of collections of objects. Furthermore, we report on some experimental results concerning the use of the various systems and the implementation of the data representations described in this paper.

1 Introduction

NoSQL datastore systems [9, 12] are a new generation of non-relational databases that support the design and development of applications that require managing persistent data, but for which traditional RDBMSs are not well suited. For example, Web 2.0 and "social" applications that require good horizontal scalability, and for which a database access based on simple operations is sufficient.

According to [13], more than fifty NoSQL systems have been already implemented, each with different characteristics (e.g., different data model and different API to access the data, as well as different consistency and durability guarantees). As pointed up in [13] this lack of standard is problematic to application developers.

To provide a better understanding of the NoSQL landscape, [9] suggested to group NoSQL data stores according to their data model in three main categories:

- *key-value stores*: systems that store values and an index to find them, based on a programmer-defined key; thus, a database is a schema-less collection of key-value pairs; e.g., Oracle NoSQL [5] and Redis [6];
- *document stores*: in these systems, a document can comprise scalar values, lists or even nested documents; there is no schema for documents, and each document can have its own attributes, defined at runtime; documents are indexed and usually a simple query mechanism is provided; e.g., MongoDB [4];
- *extensible record stores*: these systems store tables of extensible records (an extensible record is a hybrid between a tuple in a relational database and

a document in a document store) that can be partitioned across multiple nodes; e.g., Amazon DynamoDB [1], Apache Cassandra [2], and Google Bigtable [10].

In the last few years, the authors have been working on the design and implementation of two different frameworks, *SOS* (*Save Our Systems* [7]) and *ONDM* (*Object-NoSQL Datastore Mapper* [8]). Both these systems aim at exploiting the commonalities between various non-relational systems — to define a uniform application programming interface to access different NoSQL datastores in a transparent way.

This paper has the goal of sharing our experience on using several different NoSQL systems. The main contribution of this work is the description and comparison of the data models and operations offered by a number of representative non-relational systems. We consider at least a system in each of the three main categories described above. We compare models and operations of these datastores by applying them to a typical NoSQL usage scenario: the management of a database consisting of collections of objects, with the goal of providing the efficient storage and access of a single object at-a-time. Finally, we report on some experimental results, concerning the effective implementation of various data representation strategies for the NoSQL systems we consider.

A disclaimer is in order before moving forward. NoSQL systems have usually the goal of offering horizontal scalability, on the basis of a distributed implementation and deployment. They offer consistency and durability guarantees that differ from those provided by relationsl DBMSs. Most NoSQL systems support complex processing of large data sets, e.g., using MapReduce. Moreover, most of these datastores manage multiple versions of data. In this paper, we will *not* consider any of these topics, as we focus on data models and operations offered by NoSQL systems, as well as on the goal of managing collections of objects.

The paper is organized as follows: Section 2 introduces a typical NoSQL usage scenario, which defines the context for this work. Then, the data models and APIs of some representative NoSQL datastores are illustrated and compared in the following sections: Section 3 describes the key-value stores Redis and Oracle NoSQL; Section 4 presents the MongoDB document store; Section 5 discusses the extensible record stores DynamoDB and Cassandra. Section 6 illustrates and discusses some experimental results. Finally, Section 7 describes related work and in Section 8 we discuss our current work.

2 Context

In this section we describe a typical NoSQL usage scenario, that defines the context and scope of this paper. We consider a fictitious Web 2.0 or "social" application, such as an online game. The application should manage various collections of objects, including players and games. We assume that each object has a unique identifier and a complex value. These complex values can also nest the value of other objects; for example, the value of a player can nest the games she is currently playing. In this scenario, the underlying database should

```
[
   username  : "mary1994",
   firstName : "Mary",
   lastName  : "Wilson",
   games : {
              [ id : "2345", opponent : "rick_the_good", gameDetails : ... ],
              [ id : "7425", opponent : "ann x", gameDetails : ... ]
           }
]
```

Fig. 1. The complex value of a sample `Player` object

guarantee the efficient storage and retrieval of a single whole object at-a-time, given its collection and identifier. For example, when a player connects to the application, then all data concerning the player and all the games she is currently playing should be loaded in memory. Thus, the typical access is to a single object at-a-time, and requires an efficient retrieval of all the data concerning that object.

In this paper, we adopt the following terminology from [9]. A *data store* is a *system* used to store data (such as a NoSQL or even a relational database system). The term *database* refers to *data* stored in one of these systems.

The kind of data we would like to store in a NoSQL *database* is thus a set of *objects*, organized in *collections*. Each object has an *identifier* (unique within the collection it belongs to) and a *complex value*. At the top-level, the complex value of an object is a record, i.e., a set of field-value pairs. Each value can be of a simple type, a record, or a list or a set of values. The complex value of an object can be arbitrarily nested. Please note that, apart from collections, our databases are schema-less: the objects in a same collection are not required to have the same identical structure.

The operations we would like to perform *in an efficient way* are the storage and retrieval of a single object at-a-time, given its collection and identifier. We can also be interested in performing other operations (such as retrieving all the objects in a certain collection whose value satisfies a certain condition). However, we do not require that such other operations should be executed in an efficient way. This position is aligned with the kind of access operations usually supported by NoSQL datastores.

As a running example, we consider a database for storing a collection of `Player`s. Figure 1 shows the complex value of a `Player` object having username `mary1994`. An operation we would like to perform in an efficient way could be the retrieval of the complex value of a `Player` object, given its username.

In the following sections, we will describe and compare a number of representative non-relational systems. For each datastore, we will first present its data model and operations, and then discuss how to store a database for collections of objects and how to implement the access to such objects, in an efficient way, with reference to the context defined in this section.

3 Key-Value Stores

A *key-value store* is a system that stores values and an index to find them, based on a programmer-defined key, in a schema-less way. Thus, a database

is a collection of key-value pairs. Each key-value pair is a single record in the database, where the key is used to locate and access the associated value.

Several NoSQL systems belong to the category of key-value stores. As we will see, they can differ greatly in how they define what keys and values are, and in the operations they offer to access groups of key-value pairs.

3.1 Redis

Redis is "an open-source, advanced key-value store" [6]. In Redis, a *database* is a schema-less collection of key-value pairs, with a key-value index. In this section, we will first consider the basic features of Redis, and then discuss some of the features that make it an *advanced* key-value store.

Basic Data Model, Operations, and Usage. The basic Redis data type, used to define keys and values, are *binary strings*, that is, any kind of binary data, such as a byte array, a number, a plain string[1], an XML document, a serialized object, a JPEG image, or any kind of binary large object (blob). In general, Redis considers such binary strings as uninterpreted data. Thus, each key-value pair is a pair of binary strings.

As basic operations, Redis offers the following simple operations to access key-value pairs: set(*key*, *value*) adds (or modifies) the key-value pair ⟨*key*, *value*⟩ to (in) the database;[2] get(*key*) retrieves from the database the value associated with key *key*; delete(*key*) deletes from the database the key-value pair associated with key *key*. All these operations can be executed in a very efficient way, that is, essentially in constant time.

As we already said, in Redis each key is a binary string. However, it is possible (and quite common, in our experience) to use plain strings as keys. Moreover, it is also possible (and common) to give some structure to keys, for example to use keys formed as sequences of identifiers, separated by suitable separators, e.g., colons, slashes or dots. For example, *Player:mary1994/firstName*.

In Redis, values are binary strings, too. They can be used to store simple data — either in a binary or a textual representation. They can also be used to store complex values — in particular, using some serialization[3] format such as JSON [3].

On the basis of these considerations, it is possible to identify some data representation strategies for storing in Redis a database comprising collections of objects (see the context defined in Section 2).

A first strategy adopts a single key-value pair for each object (*key-value per object*, *kvpo*). The key is composed of the collection name and the identifier of the object. The value is a serialization of the whole complex value of the

[1] In this paper, we will write *plain string* to denote an ordinary string, that is, a sequence of characters, and to distinguish them from binary strings.

[2] As it is customary in many NoSQL systems, Redis offers a single operation to add or modify a database record, rather than two distinct operations.

[3] *Serialization* is the process of translating an object or a set of objects into a binary or textual format that can be stored and then reconstituted later.

key	*value*
Player:mary1994	{"username":"mary1994", "firstName":"Mary", ...}

(a) Key-value per object in Redis

key	*value*
Player:mary1994/username	mary1994
Player:mary1994/firstName	Mary
Player:mary1994/lastName	Wilson
Player:mary1994/games	...

(b) Key-value per field in Redis

key	*value*
Player:mary1994	username:mary1994 firstName:Mary lastName:Wilson games:...

(c) Key-hash per field in Redis

key	*value*
/Player/mary1994/-/username	mary1994
/Player/mary1994/-/firstName	Mary
/Player/mary1994/-/lastName	Wilson
/Player/mary1994/-/games	...

(d) Key-value per field in Oracle NoSQL

key	*value*
/Player/mary1994/-/username	mary1994
/Player/mary1994/-/firstName	Mary
/Player/mary1994/-/lastName	Wilson
/Player/mary1994/-/games/0/id	2345
/Player/mary1994/-/games/0/opponent	rick_the_good
/Player/mary1994/-/games/0/gameDetails	...
/Player/mary1994/-/games/1/id	7425
/Player/mary1994/-/games/1/opponent	ann x
/Player/mary1994/-/games/1/gameDetails	...

(e) Key-value per atomic value in Oracle NoSQL

Fig. 2. Data representation strategies for key-value stores

object. For example, consider the sample player shown in Figure 1. We can represent it using a single key-value pair, with key `Player:mary1994` and value {`"username":"mary1994", "firstName":"Mary", ...`}. See Figure 2(a).

A second strategy uses multiple key-value pairs for each object. Specifically, it adopts a key-value pair for each top-level field of the complex value of the object (*key-value per field*, *kvpf*). The key is composed of the collection name, the object identifier, and the name of the top-level field. The value is the value of the field in the specified object. For example, the sample player shown in Figure 1 can be represented using four different key-value pairs, one for each of its fields, *username*, *firstName*, *lastName*, and *games*. See Figure 2(b).

The former strategy *kvpo* above enables the efficient retrieval of an object. Indeed, a retrieval requires just a Redis get operation.

A naive implementation of the latter strategy *kvpf* can be based on an additional Redis operation, keys(*pattern*), which finds all keys matching the given *pattern*. For example, keys(*Player:mary1994/**) allows finding in the database keys such as *Player:mary1994/firstName*. However, Redis discourages the use of operation keys, for performance reasons. Moreover, the retrieval of an object would also require multiple get operations. In summary, the naive implementation of the retrieval of an object using strategy *kvpf* is not efficient.

As we will see next, Redis offers some advanced features, that make possible an efficient implementation of (a variant of) strategy *kvpf*.

Advanced Characteristics. Redis is an *advanced* key-value store, and as such it offers a number of specific features other than the ones described so far.

In particular, Redis defines a few specific data types for its values, as follows: binary strings, integer counters, lists and sets of binary strings, or hashes (a *hash* is a map between binary string fields and binary string values). It is important

to note that structured data types (i.e., lists, sets, and hashes) cannot be nested, so that, for example, it is not possible to define a list of sets of binary strings.

In correspondence to each data type, Redis defines a specific set of additional operations. For example, there are atomic operations to append a string to a binary string, to insert an element in a set or a list, or to increment a counter.

To our goals, a useful feature in Redis are hashes. A hash value can be used, in particular, to represent the various fields of an object, using a single hash for a whole object, containing a field-value pair for each of its top-level fields (*key-hash per object*, *khpo*). See Figure 2(c). Moreover, Redis offers operations to handle a whole hash in an efficient way. For example, operation hgetall(*key*) retrieves all field-value pairs associated with key *key*. Hence, data representation strategy *khpf* enables the efficient retrieval of an object.

3.2 Oracle NoSQL

Oracle NoSQL [5] is another key-value store, offering the storage of key-value pairs. Differently from Redis — which implements a simple structure for keys and an advanced organization for values — Oracle NoSQL provides the management of unstructured values on the basis of more structured keys.

In Oracle NoSQL, a *key* is composed of a *major key* and a *minor key*. The major key is a non-empty sequence of plain strings. The minor key is a possibly-empty sequence of plain strings. Each element of a key is called a *component* of the key. An example of key is `/M1/M2/M3/-/m1/m2`, where each $\texttt{M}_i$ is a major component and each $\texttt{m}_j$ is a minor component. Symbol `/` separates components, and symbol `-` separates the major key from the minor key.

On the other hand, in Oracle NoSQL a *value* is simply a binary string.

The distinction between major key and minor key is relevant with reference to the distributed implementation of Oracle NoSQL, based on sharding.[4] Indeed, key-value pairs are spread on the distributed datastores, using a hash function based only on the major key component(s). In simpler words, key-value pairs having a same major key are always allocated in a same node. Moreover, an efficient access to the key-value pairs having a same major key (and thus allocated in a same node) is provided.

The basic operations offered by Oracle NoSQL are as follows: put(*key*, *value*) adds (or modifies) the key-value pair $\langle key, value \rangle$ to (in) the database; get(*key*) retrieves from the database the value associated with key *key*; delete(*key*) deletes from the database the key-value pair associated with key *key*. All these operations can be executed efficiently, i.e., in constant time.

Moreover, Oracle NoSQL offers operations to access and manipulate a related group of key-value pairs. For example, operation multiGet(*parentKey*) retrieves multiple key-value pairs at once, provided they all share a same major key. Specifically, *parentKey* should specify a complete major key, whereas the minor key could be either omitted or partial. Operation multiGet then retrieves all key-value pairs whose key starts with the specified parent key. Since such key-value

[4] *Sharding* is the partitioning of the data in a database into multiple smaller parts, distributed across various nodes. Many NoSQL datastores adopt this technique.

pairs are always allocated in a same node, operation multiGet can be executed in an efficient way.

While Oracle NoSQL does not define a "write" counterpart of operation multiGet, it provides an execute operation for executing multiple put and delete operations efficiently — provided that the keys specified in these operations all share a same major key.

With reference to the above features, it turns out that data representation strategies *key-value per object* and *key-value per field* (described in Section 3.1) can be straightforwardly implemented using Oracle NoSQL. See, for example, Figure 2(d). The availability of the multiGet operation makes it efficient also the implementation of strategy *kvpf*.

Furthermore, Oracle NoSQL enables the implementation of another representation strategy that adopts again multiple key-value pairs for each object, but using atomic values only (*key-value per atomic value*, *kvpav*). Specifically, a key-value pair is used for each atomic value contained in the complex value of the object. The key is composed of the collection name, the object identifier, and the sequence of fields that need to be traversed to locate the specific atomic value. See Figure 2(e) for the representation of the sample player shown in Figure 1.

4 Document Stores

A *document store* is a system that store documents, where a document is essentially a complex value, which can comprise scalar values, lists, and nested documents.

4.1 MongoDB

MongoDB [4] is an open-source, document-oriented data store that offers a full-index support on any attribute, a rich document-based query API and Map-Reduce support.

In MongoDB, a *database* comprises one or more collections. Each *collection* is a named group of documents. Each *document* is a structured document, that is, a complex value, a set of attribute-value pairs, which can comprise simple values, lists, and even nested documents. Thus, documents are neither freeform text documents nor Office documents. Documents are schema-less, that is, each document can have its own attributes, defined at runtime.

Specifically, MongoDB documents are based on BSON (Binary JSON), a variant of the popular JSON format. Values constituting documents can be of the following types: *(i) basic types*, such strings numbers, dates, and boolean values; *(ii) arrays*, i.e., ordered sequences of values; *(iii) documents* (or *objects*): a document is a collection of zero or more key-value pairs, where each key is a plain string, while each value is of any of these types.

MongoDB documents can be nested using either the *document embedding* feature, that saves a document inside another one, or the *document linking* feature, that links nested documents by means of references.

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

collection	*document id*	*document*
Player	mary1994	{"_id":"mary1994", "username":"mary1994", "firstName":"Mary", ...}

(a) Document per object in MongoDB

table	*_id*	*username*	*firstName*	*lastName*	*games*
Player	mary1994	mary1994	Mary	Wilson	{ ... }

(b) Item per object in DynamoDB

table	*id*	*value*
Player	mary1994	{"username":"mary1994", "firstName":"Mary", ...}

(c) Cell per object in Cassandra

Fig. 3. Data representation strategies for document and extensible record stores

The basic operations offered by MongoDB are as follows: insert(*coll*, *doc*) adds a main document *doc* into collection *coll*; find(*coll*, *selector*) retrieves from collection *coll* all main documents matching document *selector*. The simplest selector is the empty document {}, which matches with every document; it allows to retrieve all documents in a collection. Another useful selector is document {*_id:ID*}, which matches with the document having identifier ID. Operation remove(*coll*, *selector*) removes from collection *coll* all documents matching document *selector*.

A data representation strategy that can be implemented in MongoDB adopts a single main document for each object (*document per object*, *dpo*). Each distinct collection of objects is stored in a separate MongoDB collection. An object having identifier ID and value V is represented by a document whose value is a serialization of V and that includes an additional key-value pair {*_id:ID*}. See Figure 3(a).

5 Extensible Record Stores

An *extensible record store* is a datastore organized around tables, rows, and columns. Databases are mostly schema-less, since each row can have its own set of columns. In a distributed implementation, rows and columns are used to shard data over multiple nodes. These datastores are also called *column-family stores* [12].

5.1 DynamoDB

Amazon DynamoDB [1] is a NoSQL database service provided on the cloud by Amazon Web Services (AWS), which focuses on high availability, flexibility, and scalability. It can be classified as an extensible record store.

In DynamoDB, a *database* is organized in tables. A *table* is a set of items. Each *item* contains one or more *attributes*, each with a *name* and a *value* (or a set of values). Each table should designate an attribute as *primary key*. Then, each item in the table is required to have a unique value for the 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 database are mostly schema-less.

DynamoDB indexes data only with respect to primary-key attributes. Specifically, a primary key is composed of a *hash partition attribute* and an optional *range attribute*. Sharding is based only on the partition attribute.

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

A data representation strategy that can be implemented in DynamoDB adopts a single item for each object (*item per object*, *ipo*). Each distinct collection of objects is stored in a separate table. Each table has an attribute designed as primary key, say, *_id*. Then, an object O in a collection C having identifier ID and value V is represented by an item in the table for C, with primary key ID, having a distinct attribute for each top-level field f of O, whose value is the value of f in V (or a serialization of such value, thereof). See Figure 3(b). This strategy can be implemented in efficient way, since a whole object can be retrieved using a single getItem.

5.2 Apache Cassandra

Apache Cassandra [2] is another extensible record store. Its data model is based on tables, rows, and columns. Similarly to a relational database, in Cassandra a database consists of a set of *tables* (which are also called *column families*). Each table is organized in *rows* and *columns*. Each row in a table has a unique *row key*. The intersection of a row and a column is called a *cell*; each cell stores an uninterpreted binary string. Differently from a relational database, each row in a table is not required to have the same set of columns. Moreover, the structure of a Cassandra database is dynamic, since columns need not to be specified in advance. In practice, each table stores a sparse data set, since only some of the cells are populated.

Data access is based on simple read/write operations, each over a table and via a row key. Data access can be either a row at a time, or a cell at a time.

The data model of Cassandra is similar to that of DynamoDB, even if the terminology is quite different. Indeed, Cassandra tables, rows, row keys, and columns essentially correspond to DynamoDB tables, items, primary keys, and attributes.

In Cassandra we can implement a *row per object* (*rpo*) data representation strategy — analogous to strategy *item per object* in DynamoDB (described in Section 5.1).

Another representation strategy adopts a single cell for each object (*cell per object*, *cpo*). Each object is stored in a distinct row, having two columns: *id* for the object identifier and *value* for a serialization of the whole complex value of the object. See Figure 3(c).

6 Experimental Results

The various data representation strategies described in this paper have all been implemented in a working prototype, which is able to access the representative

Datastore	*Strategy*	*(a)*	*(b)*	*(c)*	*(d)*	*(e)*
Redis	kvpo	1.60	5.63	2.57	0.65	0.89
Redis	kvpf	1.71	6.57	3.71	0.60	1.11
Redis	khpo	1.51	6.25	3.51	0.59	1.26
Oracle NoSQL	kvpo	1.68	6.94	2.87	0.90	1.08
Oracle NoSQL	kvpf	2.25	9.67	4.91	1.02	1.75
Oracle NoSQL	kvpav	2.84	24.61	24.88	3.56	9.88
MongoDB	dpo	2.17	7.85	3.35	0.75	1.02
Cassandra	cpo	4.01	17.10	5.17	2.16	1.95
Cassandra	rpo	3.60	17.38	6.33	1.62	2.19

(a) Baseline performances (*sec*)

Datastore	*Strategy*	*(a)*	*(b)*	*(c)*	*(d)*	*(e)*
Redis	kvpf	1.58	1.67	1.74	1.78	2.29
Oracle NoSQL	kvpf	2.07	1.78	2.05	1.69	1.91
Cassandra	rpo	3.52	2.76	3.11	2.74	4.34

(b) Relative penalty due to naive implementations (*penalty factor*)

Fig. 4. Experimental results

NoSQL systems we have described so far. In this section be briefly report on the performances of the different systems and strategies.

Our experiments refer to a database for customers and orders. Each customer has 9 top-level fields, and 11 atomic values. Each order has 11 top-level fields, and an average of 28 atomic values.

Figure 4(a) shows the baseline of our experimental results. Each row refers to a specific combination datastore-data representation strategy.[5] The five columns list the timings (in seconds, and the least significant digit is hundredths of a second) with respect to five different workloads, as follows: *(a)* creation of 5000 new customers; *(b)* creation of 10000 new orders (each also requires the retrieval of a customer); *(c)* retrieval of 5000 orders (each also requires the retrieval of the associated customer); *(d)* update of a top-level field of 2000 customers (each also requires the retrieval of the customer); *(e)* update of a top-level field of 2000 orders (each also requires the retrieval of the order and its customer).

We can draw the following conclusions, based on a comparison of timings, rather than to an analysis of absolute timings.

For key-value stores, when the workload is just creation of objects and retrieval of objects (workloads *(a)-(c)*), strategy *kvpo* outperforms strategies *kvpf* (slightly) and *kvpav* (significantly). However, strategy *kvpf* can be better then *kvpo* when we need to update single fields (workload *(d)*), even if this is not true in general (workload *(e)*). The use of specific data types (such as hashes in Redis, strategy *khpo*) can be beneficial, even if this is not always true (compare rows referring to Redis *khpo* and Redis *kvpf*).

We can draw similar conclusions for extensible record stores, if we compare strategies *cpo* (analogous to *kvpo*) and *rpo* (analogous to *kvpf*).

[5] We do not report on DynamoDB, since its runs on the cloud and therefore timings are not easily comparable with timings for datastores which run on a local data server.

Furthermore, in Figure 4(b) we compare the use of efficient multi-put and multi-get operations with the use of naive implementations based on multiple simple put and get operations.[6] The numbers in the table report the penalty timing factors one should pay when using naive implementations. For example, 1.58 means that the corresponding timing is 1.58 times the baseline performance reported in Figure 4(a). As the table reports, in our experiments this penalty ranges from 1.50 to more than 4.

In summary, these experiments suggest that the adoption of a NoSQL datastore requires a deep understanding of all the following aspects: *(i)* the data model offered by the datastore; *(ii)* the possible representation strategies enabled by such data model; *(iii)* the data access operations offered by the system; and *(iv)* the efficiency of such operations.

7 Related Work

According to Stonebraker [13], more than fifty NoSQL datastores have been implemented. `http://nosql-database.org/` lists about 150 non-relational database systems. Each datastore has its own documentation — which is often difficult to understand and sometimes even incomplete. The documentation of a few systems discuss some useful guidelines and best practices for obtaining the best performances and results from that datastore. With this huge amount of heterogeneous information, it is difficult to have a clear vision of the NoSQL landscape.

A survey by Cattell [9] describes a number of NoSQL datastores. It offers an overview of this technology. It also provides a useful classification of non-relational systems. However, it is not focused on data models and APIs, which are described only in general terms. Moreover, it does not consider methodological aspects concerning data modeling. On the other hand, in this paper we focus on the data models and APIs of a number of representative NoSQL systems. We also consider some methodological aspects related to data modeling.

A few systems have been proposed with the aim of hiding the heterogeneity of NoSQL datastores and to provide a uniform access to them. In particular, SOS [7] and ONDM [8], which we will briefly describe in the next section.

8 Discussion

This work has the goal of sharing our experience on using several NoSQL datastores while working on the design and implementation of two different frameworks, SOS and ONDM. Both these systems have the goal of defining a uniform application programming interface to access different NoSQL datastores in a transparent way.

SOS (*Save Our Systems* [7]) is a platform that provides a uniform interface towards NoSQL systems. It exposes operations for storing, retrieving, deleting,

[6] Strategy *kvpo* uses simple put and get operations only, so the naive implementation is the more efficient. We do not have realized the naive implementations for all strategies, and therefore we do not report on, e.g., strategy *kvpav* in Oracle NoSQL.

and modifying data — they are inspired by the ones offered by the underlying datastores. SOS hides specific features of NoSQL systems, thus allowing programmers to remain unaware of the individual characteristics and query patterns of each datastore. The use of SOS provides also interoperability. Indeed, an application can benefit from the usage of multiple NoSQL datastores in a transparent way, by simply accessing them through the same interface. The implementation of SOS is based on a metamodel approach that maps the specific datastore interfaces into a common and general one. Mappings specify how to store and query data, and are defined between the metamodel and each system.

ONDM (*Object-NoSQL Datastore Mapper* [8]) is a framework for facilitating the storage and retrieval of persistent objects in NoSQL datastores, which aims at supporting several challenges posed to application developers by the heterogeneity in the NoSQL arena. The key features of ONDM are as follows: *(i)* ONDM offers to application developers an ORM-like API, based on the popular Java Persistence API (JPA [11]). As JPA, it is based on an entity data model, with entities, relationships, and embeddable objects (i.e., complex values). *(ii)* ONDM already supports the access to a handful of NoSQL systems (Apache Cassandra, Couchbase, Oracle NoSQL, MongoDB, and Redis), belonging to different datastore categories. More important, ONDM has been designed to be easily extensible. *(iii)* For each NoSQL datastore, ONDM implements multiple entity representation strategies (such as the ones described in this paper). Moreover, ONDM has been designed to easily incorporate new entity representation strategies. *(iv)* Using ONDM, it is possible to store relationships between objects using two distinct relationship representation modes: as references (i.e., normalized) or materialized (i.e., denormalized).

References

1. Amazon DynamoDB. `http://aws.amazon.com/dynamodb`. Accessed 2013.
2. Apache Cassandra. `http://cassandra.apache.org`. Accessed 2013.
3. JSON. `http://www.json.org`. Accessed 2013.
4. MongoDB. `http://www.mongodb.org`. Accessed 2013.
5. Oracle NoSQL Database. `http://www.oracle.com/technetwork/products/nosqldb`. Accessed 2013.
6. Redis. `http://redis.io`. Accessed 2013.
7. Paolo Atzeni, Francesca Bugiotti, and Luca Rossi. Uniform access to non-relational database systems: The SOS platform. In *CAiSE 2012*, pages 160–174, 2012.
8. Luca Cabibbo. ONDM: an Object-NoSQL Datastore Mapper. Submitted for publication, 2013.
9. Rick Cattell. Scalable SQL and NoSQL data stores. *SIGMOD Record*, 39(4):12–27, 2010.
10. Fay Chang et al. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26(2), 2008.
11. Java Persistence 2.0 Expert Group. The Java Persistence API 2.0, jsr 317, 2009.
12. Pramodkumar J. Sadalage and Martin J. Fowler. *NoSQL Distilled.* Addison-Wesley, 2012.
13. Michael Stonebraker. Stonebraker on NoSQL and enterprises. *Commun. ACM*, 54(8):10–11, 2011.