Information Systems ▮ (▮▮▮▮) ▮▮▮–▮▮▮

Contents lists available at SciVerse ScienceDirect

Information Systems

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

Uniform access to NoSQL systems

Paolo Atzeni *, Francesca Bugiotti, Luca Rossi

Università Roma Tre, Via della Vasca Navale 79, 00146 Roma, Italy

ARTICLE INFO

Keywords:
Non-relational databases
NoSQL
Interoperability

ABSTRACT

Non-relational databases (often termed as NoSQL) have recently emerged and have generated both interest and criticism. Interest because they address requirements that are very important in large-scale applications, criticism because of the comparison with well known relational achievements. One of the major problems often mentioned is the heterogeneity of the languages and of the interfaces they offer to developers and users. Different platforms and languages have been proposed, and applications developed for one system require significant effort to be migrated to another one. Here we propose a common programming interface to NoSQL systems called SOS (Save Our Systems). Its goal is to support application development by hiding the specific details of the various systems. It is based on a metamodelling approach, in the sense that the specific interfaces of the individual systems are mapped to a common one. The tool provides interoperability as well, since a single application can interact with several systems at the same time.

1. Introduction

Relational database systems (RDBMSs) dominate the market by providing an integrated set of services that refer to a variety of requirements, which mainly include support to transaction processing but also refer to analytical processing and decision support. From a technical perspective, all the major RDBMSs on the market show a similar architecture (based on the evolutions of the building blocks of the first systems developed in the Seventies) and do support SQL as a standard language (even though with dialects that differ somehow). They do provide reasonably general-purpose solutions that balance the various requirements in an often satisfactory way.

However, some concerns have recently emerged towards RDBMSs. First, it has been argued that there are cases where their performances are not adequate, while dedicated engines, tailored for specific requirements (for example decision support or stream processing) behave much better [20] and provide scalability [19]. Second, the structure of the relational model, while being effective for many traditional applications, is considered to be too rigid or not useful in other cases, with arguments that call for *semistructured* data (in the same way as it was discussed since the first Web applications and the development of XML [1]). At the same time, the full power of relational databases, with complex transactions and complex queries, is not needed in some contexts, where "simple operations" (reads and writes that involve small amount of data) are enough [19]. Also, in some cases, *ACID* consistency, the complete form of consistency guaranteed by RDBMSs, is not essential, and can be sacrificed for the sake of efficiency. It is worth observing that many Internet application areas, for example, the social networking domain, require both scalability (indeed, Web-size scalability) and flexibility in structure, while being satisfied with simple operations and weak forms of consistency.

With these motivations, a number of new systems, not following the RDBMS paradigm (neither in the interface nor in the implementation), have recently been developed. Their common features are scalability and support to simple operations only (and so, limited support to complex ones), with some flexibility in the structure of data. Most of them

* Corresponding author. Tel.: +39 0657333213.
E-mail addresses: atzeni@dia.uniroma3.it (P. Atzeni), franbugiotti@yahoo.it (F. Bugiotti), luca@rossi.net (L. Rossi).

0306-4379/$ - see front matter © 2013 Elsevier Ltd. All rights reserved.
http://dx.doi.org/10.1016/j.is.2013.05.002

also relax consistency requirements. Some of them can be deployed and managed on local servers, while others offer their services on the cloud as data services [9]. They are often indicated as *NoSQL* systems, because they can be accessed by APIs that offer much simpler operations than those that can be expressed in SQL. Probably, it would be more appropriate to call them *non-relational*, but we will stick to common usage and adopt the term NoSQL.

There is a variety of systems in the NoSQL arena [10,19], more than fifty, and each of them exposes a different native interface (different data model and different API). Indeed, as it has been recently pointed out, the lack of standard is a great concern for organizations interested in adopting any of these systems [18]: applications and data are not portable and skills and expertise acquired on a specific system are not reusable with another one. Also, each of these systems has specific goals, and so it is tailored on one or few specific usage patterns. As a consequence, it might be possible that complex applications could benefit from the use of several NoSQL systems at the same time, and in this case the heterogeneity causes even bigger difficulties.

The observations above have motivated us to look for methods and tools that can alleviate the consequences of the heterogeneity of the interfaces offered by the various NoSQL systems and can also enable interoperability between them, together with ease of development (by improving programmers' productivity, pursuing one of the original goals of relational databases [14]).

As a first step in this direction, we present here *SOS* (*Save Our Systems*), a programming environment where different non-relational databases can be uniformly defined, queried and accessed by an application program.

The programming model is based on a high-level common interface, which is inspired by those of non-relational systems, but remains general with respect to their specific features, being adaptable to a large number of systems and models.

The common interface allows us to pursue a two-fold objective:

- Developing applications that are generic with respect to the underlying NoSQL system, thus obtaining a form of "physical independence," that resembles the well known one for relational databases [13]: code would refer to a "logical" level (our interface) and our tool would handle the interaction with the actual "physical" data store. As a consequence, applications would be portable and reusable, as the same high level code works for different systems.
- Enabling uniform access to diverse data stores. This would apply also to data previously managed by applications using native interfaces.

In some sense, we can say that the first objective is "top-down" as it starts from applications and generates the implementation of the data in the specific systems, whereas the second is "bottom-up" as it allows to access data already managed by the specific systems.

Following the common features of NoSQL systems, our interface supports operations (insertion, update, retrieval) on individual objects and, in some cases, sets thereof. Objects are allowed to have a hierarchical nested structure, again motivated by the features of the systems, and their retrieval is based on a path language, which exploits the hierarchical structure. We have developed techniques to handle the differences between the various systems, with instantiation (indeed, implementation) in each of them.

We have experimented with various systems and, in this work, we will discuss implementations for three of them with rather different features within the NoSQL family: namely, Redis,[1] MongoDB,[2] and HBase.[3] Indeed, the implementations are transparent to the application, so that they can be replaced at any point in time (and so one NoSQL system can be replaced with another one). Also, our platform allows for a single application to partition the data of interest over multiple NoSQL systems, and this can be important if the application has contrasting requirements, satisfied in different ways by different systems.

To the best of our knowledge, the programming model we present in this paper is original, as there is no other system that provides a uniform interface to NoSQL systems. It is also a first step towards a seamless interoperability between systems in the family, where code written for a given system would be enabled to access other systems.

The rest of this paper is organized as follows. In Section 2 we briefly present the major families of NoSQL systems and comment on the three specific ones we discuss in the implementation. In Section 3 we introduce a running example that will be used throughout the paper. In Section 4 we illustrate the common interface we propose. In Section 5 we discuss how the data model in the common interface can be implemented in the various underlying systems. In Section 6 we show how the operations and the path language in the common interface can be mapped to those of the specific systems. In Section 7 we present the implementation of our tool. In Section 8 we illustrate the development of the running example in our platform. In Section 9 we report on some experiments we conducted on the tool. Finally, we briefly discuss related work (Section 10) and draw our conclusions (Section 11).

2. NoSQL systems and their data models

Most NoSQL systems (also called NoSQL *data stores*, to avoid confusion with the term *database system*, which often indicates traditional database systems [10]) have been developed independently from one another, each with specific application objectives, but with the general goal of offering rather simple operations on flexible data structures. Indeed, they usually leverage on this simplicity to provide high scalability and massive throughput. A feature that is common to almost all systems (and coherent with the principle of simple operations) is that they handle individual items, identified by unique keys. Systems differ on the structure that is supported for these individual

[1] http://redis.io
[2] http://www.mongodb.org
[3] http://hbase.apache.org

Please cite this article as: P. Atzeni, et al., Uniform access to NoSQL systems, Information Systems (2013), http://dx.doi.org/10.1016/j.is.2013.05.002

items, on the type constructors (map, set, list) that can be used, and on the possibility of nesting. Also, structures are flexible, in the sense that "schemas" are often relaxed or completely absent.

Being this a relatively young space, consolidated standards are yet to be found, and, given both the number of NoSQL data stores and the differences between them, it is useful to group them in categories, according to some criterion. An interesting classification on data modelling features has been recently proposed [10]; it groups systems into three major families: extensible record stores, document stores and key-value stores. Systems belonging to the same family largely agree on main data structures and access patterns, whereas they may differ in specific operations support, structure details and in architectural aspects like consistency models, partitioning, and so forth.

In the rest of this section we will describe in detail these three major NoSQL families by means of their main features and referring to a representative system for each of them. Also, being the focus of our work on data modelling themes, we will concentrate on aspects like data structures, schema features and access patterns.

2.1. Extensible record stores

Let us start with extensible record stores, whose data model shows some apparent similarity with the relational one, and so it can be described in terms of differences with respect to it. In an extensible record store, a database is composed of a set of collections, which are called *tables*, as in the relational model, but have indeed a much more flexible structure, with columns that are not predefined, and so rows are often "sparse." At the same time, columns are grouped into *families*, for vertical partitioning and so as a basis for physical optimization.

The data stores in this family all originate from BigTable [11], a system developed by Google for internal use and not made available outside the company. Here we refer to HBase, which is modelled after BigTable, but openly available.

In the same way as BigTable, HBase has tables, whose names are unique within a given database instance, and columns, grouped into families.

Columns are not predefined, and so can be defined dynamically, at insertion or modification time. Indeed, the creation of new tables and new column families is not possible during "normal operations" and has to be done only at "schema design time," while the data cannot be accessed. Therefore, we can say that, while these systems do not require a fixed, predefined schema, they are not really "schemaless," as some schema elements have to be defined in advance. Column families are used as the basis for physical partitioning, and the documentation specifies that it is better to keep the number of families rather low.

Columns are denoted by the name of the family together with the name of the column in the family, often called *qualifier* (this is often denoted in the form "*family: qualifier*," however methods have separate arguments for them). This means that column qualifiers need to be unique within each family, but can be reused in different families (thus denoting different columns). In other words, the column family is a namespace for column qualifiers.

Tables contain *rows*, each of which has a key that uniquely identifies it (within the table) and one or more columns with values (each within a column family). The structure offered by HBase is often referred to as a *multidimensional*, *sparse map*. It is *multidimensional* because each value is identified by a composed key: the row key, the column family, the column, and also a *timestamp*, used for versioning purposes. The map is also *sparse*, because each row has usually values only for some of the columns, often very few with respect to all those defined for the various rows in a table.

The value of a row for a certain column is an uninterpreted byte array, so it has no additional structure, but, obviously, it could be used to store a serialized value for a complex object.

Rows are added to tables by specifying a value for the key and for at least one of the columns. A single operation may insert one row or (for performance reasons) a set (indeed, a list) of them.

In terms of retrieval operations, HBase supports table scanning (on the basis of the row key order, as the physical arrangement of rows is sorted on keys) as well as direct access (*get* operation) on the basis of the key. Moreover, it provides *filters*, which allow for the specification of conditions for the rows and values to be retrieved (by a scan or get). Filters are executed on the server, so they really perform selections on the data store.

In Fig. 1 we show an HBase table, User, with the typical conventions for the systems in this family. Column families are emphasized in the layout: we have *Account*, *Personal* and *Friends*. The *Account* column family stores information about the user account with columns like *username* and *password*. The *Personal* column family stores personal details, while *Friends* stores friendships data. Each friend is associated with a number of columns, which all have the user identifier as a prefix, and store contact information such as name and email. The column names (with the identifier as prefix) in the *Friends* column family show a

User			
	Account	Personal	Friends
1001	username = "bob1987" password = "thisisapassword"	firstName = "Bob" lastName = "Smith" ssn = "4hfe94"	2004:firstName = "Alice" 2004:lastName = "Smith" 2004:email = "alice@gmail.com" 1714:firstName = "Charlie" ...
2004	username = "alice"	...	...
1714	...	...	...

Fig. 1. An HBase table.

Please cite this article as: P. Atzeni, et al., Uniform access to NoSQL systems, Information Systems (2013), http://dx.doi.org/10.1016/j.is.2013.05.002

common practice to handle sets of values and references to other rows. However, the practice should be enforced by the applications that store data and it is not supported by the system.

2.2. Document stores

Document stores handle collections of objects (called *documents*) represented in hierarchical formats such as XML or JSON. Each document is composed of a (nested) set of fields and is associated with a unique identifier, for indexing and retrieving purposes. Generally, these systems offer a richer query language than those in other NoSQL categories, being able to exploit the structuredness of the objects they store. Among document stores we refer to MongoDB, one of the most adopted.

In MongoDB a database instance contains a set of collections, each identified by a name (so, we could say that a database instance is a map of name-collection pairs). Collections are not predefined, in the sense that a collection with a given name is created when a document is first inserted into the collection with that name. Collections have no limit in cardinality, nor any constraint on the structure (fields and nesting) of the documents they contain. The only regularity is in the fact that each document has a key (*_id*) which is unique within the collection it belongs to. Therefore, documents are identified by means of the name of the collection together with the *_id* value. Given the flexibility in the existence of collections and on the structure of the documents in them, we can definitely claim that MongoDB follows a schemaless approach. In practice, it is common to use a collection to store objects that have the same semantics and similar structure, but this is not a constraint and the flexibility turns out to be very important in many applications, especially with respect to the evolution of requirements.

Specifically, MongoDB documents are represented in BSON (Binary Serialized dOcument Notation), a binary serialization of JSON, with which it shares the schemaless, arbitrarily nested structure. BSON (just in the same way as JSON) defines three major element types for document fields: *object*, *array*, and *value*. An object is a map made of key-value pairs, where values can belong to any element type. Arrays are ordered collections of elements (with no constraint on structure), while values are simple data types such as strings or integers.

Documents within a collection can be sequentially scanned, individually retrieved using the identifier or queried according to a pattern that matches the structure of the fields. It is also possible to add new documents into a collection at any time.

As an example, let us consider in Fig. 2 the same scenario as we used for HBase: a collection of users having some account data, a set of friends, and a structured set of personal info. According to the modelling structure, data would be organized in separate documents (one for each different person) identified by a key, within a same collection (named, for instance, *users*) The personal data can be represented as an object while friends are collected into an array structure (denoted by square brackets).

2.3. Key-value stores

Key-value systems are somehow the most distant from relational databases, because they do not have any notion of schema, not even collections. In the systems of this category, the whole database is essentially a map: it contains objects (called *values*) each identified by a unique *key*. The various systems differ in the form that keys might have and in the types of the values, and so they can be more or less sophisticated. At the same time they share the idea that insertion, retrieval, and removal refer to single objects and are specified by indicating the key. Operations spanning multiple objects are often not trivial or not supported at all. Values can be simple elements such as strings and integers, or structured objects, depending on the expressive power of the specific system. In the systems that handle only simple types, serialization of complex objects in values is in general possible, but operations cannot take advantage of the structure. With respect to keys, some systems consider them just as strings, while others allow for a hierarchical structure, composed of various portions. However, even when keys are just strings, it is common (as we will see shortly) to use conventions that mimic hierarchical structures.

```
users: [
    {
        _id: "1001",
        username: "bob1987",
        password: "thisisapassword"
        personal: {
            firstName: "Bob",
            lastName: "Smith",
            ssn: "4hfe94"
        },
        friends: [
            {
                id: "2004",
                firstName: "Alice",
                lastName: "Smith",
                email: "alice@gmail.com"
            },
            {
                id: "1714",
                firstName: "Charlie",
                ...
            },
            ...
        ]
    },
    {
        _id: "2004",
        username: "alice",
        ...
    }
    {
        _id: "1714",
        ...
    }
    ...
]
```

Fig. 2. A MongoDB collection.

We chose Redis as a representative of key-value data stores. It is rich in terms of data structures and operations (and so allows for the demonstration of various features) and at the same time simple in the structure of keys (and so it requires the adoption of interesting techniques).

A Redis database is indeed a single key-value map, where keys are strings and values may belong to different data types: string, list, set, sorted set, and hash. *Lists* represent ordered collections of strings. *Sets* are unordered collections of strings, not allowing duplicate elements. *Hashes* are maps (in other words, records), that is, values identified by keys, where both keys and values are strings. Finally, *sorted sets* are ordered collections of strings, without duplicates, where each element is also given an explicit, numeric value, which is used to keep the structure sorted. It is important to observe that the types cannot be composed (the components are always strings) so nested values are not managed (at least not directly, as we will see shortly that indirect ways do exist).

As in programming languages, each of these data types has in Redis a number of native operations that can be used to access, insert and retrieve data. For instance, lists can be accessed by conventional methods like *insert*, *remove* and *length*, but also by methods like *push* and *pop* for simulating specific structures like *stacks* or *queues*.

In terms of schema support, Redis is completely unstructured: there is no notion of predefined collections, nor any other logical or physical feature visible to the user. All the operations (including those of configuration and maintenance) are meant to be executed at any time, with no performance penalty.

As it turns out, the diversity of structures and operations makes Redis a highly flexible data store, where each modelling scenario can be accomplished in many alternative ways. Indeed, a number of conventions and best practices have been adopted in order to simplify the development of common use cases. Among these, one of the most common is the use of *descriptive*[4] keys, that is to say, keys composed of a number of elements, with conventional separators, which describe the complex structure. The goal is to create hierarchies and different key spaces, which are not supported natively in Redis. Let us consider an example:

```
users:1001:firstName="Bob"
users:1001:lastName="Smith"
```

The code shows two descriptive keys, with the pattern (which is indeed common) *collection*:*id*:*field*. Here, *collection* denotes a group of homogeneous objects, *id* is the unique identifier of the object within the collection, while what follows is the actual name of the field we are representing. Possibly, fields following the *id* can be arbitrarily nested (and so concatenated to form a complex key), allowing suitable representations for potentially complex structures, as follows:

```
users:1001:friends:2004:email="alice@gmail.com"
```

However, if not used wisely, these practices can lead to overly complicated key structures which tend to be rigid, redundant, and hard to maintain. For this reason, simple fields are usually grouped into hashes, which provide an effective way to model objects, while keeping the structure dry, as in the next example.

```
users:1001={
  username="bob1987"
  password="thisisapassword"
}
```

Since in Redis hashes cannot be nested (their values must be strings), descriptive keys have still an edge at providing, when needed, artificial hierarchies, as follows:

```
users:1001={
  username="bob1987"
  password="password"
  friends:2004.email="alice@gmail.com"
  friends:2004.firstName="Alice"
  ...
  }
```

3. Running example

In order to show how our proposed system can support application development, in this paper we refer to an example regarding the definition of a simplified version of Twitter,[5] the popular social network application. Transactions are short-lived and involve little amount of data, so the adoption of NoSQL systems is meaningful.

The data of interest for the example have a rather simple structure, sketched in Fig. 3: we have users, who write posts; every user "follows" the posts of a set of users and can, in turn, "be followed" by another set of users. As it is common in many applications, the main objects are somehow predefined, and so we can even say that there is a "schema" for them, as shown in the figure. However, the objects themselves do not have a rigid structure: this is indeed the case in many applications in the Web arena, which have to satisfy evolving requirements. We assume that users have structured addresses, but the specific structure of each address is not predefined; for example, there can be simple predefined fields (such as street, number, and post-code) or some personalized labelled fields. Similarly, we assume that the user has some login and personal information (such as name, surname, title), again with no specific, predefined structure.

In the rest of the paper the specific characteristics of our model and query procedure will be illustrated with reference to the example, and then, in Section 8, we will show how SOS can support an implementation of the application using three different NoSQL data stores, saving in each one different subset of the data of the example.

[4] These are sometimes called "intelligent" keys, but we prefer to avoid such a term, as in databases it is used to refer to key values that encode content (and so they are criticized), while here they describe the structure where the content appears.

[5] http://twitter.com/

Please cite this article as: P. Atzeni, et al., Uniform access to NoSQL systems, Information Systems (2013), http://dx.doi.org/10.1016/j.is.2013.05.002

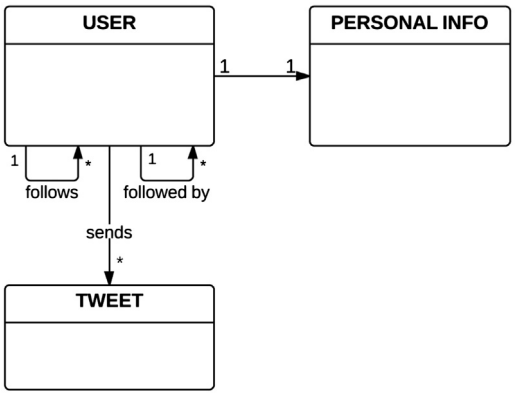

Fig. 3. The schema for the data in the example.

This is because we assume that quantitative application needs have led the software architect to drive the decision towards the use of several NoSQL DBMSs, because the various components of the application can benefit each from a different system.[6]

4. The common interface

As we said in the Introduction, the goal of our approach is to offer a uniform interface that would allow access to data stored in different NoSQL systems, without knowing in advance the specific one, and possibly using different systems within a single application. In this section we discuss the desirable features of such an interface and then present our proposal for it. In the next sections we will then describe how this interface can be implemented in various systems and the underlying architecture that allows for accessing data stored in NoSQL systems.

As we saw in Section 2, NoSQL systems are based on simple operations for inserting and deleting individual items (objects in an application), mainly one at the time, and retrieving them, one at the time or a set at the time. The individual items do have some structure, which is however not fixed in advance, apart from some limited features (names of collections, the first level of structure). Therefore, a first significant common feature of the various systems is that they mainly manipulate one object at the time. More precisely, this is almost uniformly the case for update (insertion, deletion, modification) operations, while for retrieval operations there are some differences, as we saw in Section 2.

As a consequence, it is pretty natural to define an interface that would be meaningful for the various systems of interest by means of a set of very basic and general operations on objects, PUT, GET and DELETE.

Now, the real goal of the interface is to be common to the various systems but at the same time to be able to exploit the major specific features of each of them. Therefore, there is a need to better specify the various operations, with respect to (i) the nature and structure of the objects and their organization in collections, and (ii) how many objects are involved and how they are specified.

In the next two subsections we consider these two issues in turn.

4.1. Collections, objects, and their structure

Let us first concentrate on the nature of objects. Indeed, a simple way to proceed here would be to find the "greatest lower bound" of the structuring features of the various systems: this would mean to ignore the structure of objects, as there are systems that just handle key-value pairs, without any interest in the structure of the value, and also to neglect collections, as some systems do have them and others do not. It is clear that this "minimalistic" approach would not be much effective, as it would ignore all the specificity of the various systems, reducing the interface to a bare intersection of features, very poor indeed.

Instead, by observing the systems, it turns out that it could be meaningful to expose both collections and objects with structure. In fact, if a system does not handle collections, then all objects could be held together (possibly with some use of descriptive keys to keep track of collections, when specified). Similarly, if structure is not handled in full, then the non-manageable part can be serialized.

Specifically, our interface allows to operate (put, get, etc.) on objects that have a hierarchical structure and belong to collections (in the sense that the interface allows to refer to named collections). In other words, we have a "data model" for our interface (referred to in the following as the *common data model*), which exposes collections of nested objects following the same approach of some document store systems. Let us comment on the various aspects in turn.

Each collection has a name, and individual operations refer to collections by using such a name: so, they insert objects into a specific collection, and so on. In a sense, collections correspond to a light form of schema, as collections are visible, in the same way as tables are defined and visible in relational databases. There is a number of reasons for which we say that this is a light-weight approach to schemas. First of all, collections need not be defined explicitly, as opposed to what happens in relational systems, where a CREATE TABLE statement is needed before a SELECT on the same table is issued: here, in line with what happens with many NoSQL systems, a command that refers to a non-existing collection would initialize it. Second, the internal structure of collections (and of objects in them) is not predefined. Third, collections are optional (in which case we assume operations refer to a "default" collection).

Let us now illustrate objects. Each object has a key, which is unique (and so allows for identification) within the collection of interest. The notion of key is common to all NoSQL systems and so this is a pretty natural feature. Objects have a nested structure, based on three main constructs: *struct* (also referred to as *record* or *map*), *set*

[6] For the sake of space here the example has to be simple, and so the choice of multiple systems is probably not justified. However, as the various systems have different performances and different behaviour in terms of consistency, it is meaningful to have applications that are not satisfied with just one of them.

and *attribute*, which are well known for being at the basis of complex-object models or nested relational ones with arbitrary nesting (as discussed in textbooks [3,7] and also used in pieces of work that need a general structure for semistructured data [2,12,17]).

As usual in these settings, *attributes* are the basic elements of the model, corresponding to simple values such as strings or integers. Structs and sets, instead, are complex elements whose values may be both attributes and sets or structs as well. As usual, a *struct* corresponds to a map (a record, in more traditional terminology) whose elements, identified by a name (called *key* in maps and *field* name or *attribute* for records), can be simple or be structured or sets in turn. Finally, sets are used to handle groups of elements, mainly records, but also attributes or further sets, or even heterogeneous elements. In practice, we would expect sets to be rather homogeneous, but this is not a constraint.

Incidentally, it can be observed that even collections could be handled within the nested structure of objects, because, when there are collections, we could see the whole database as a struct, with a field for each collection, and then each collection would be a set, and its elements would be usually structs. However, we decided to make collections explicit, as in this way they can be directly referenced in the calls to the SOS interface.

With respect to objects, our common model is assumed to be schemaless: objects in collections have an arbitrary hierarchical structure made of sets, structs, or attributes as well, in no predefined way. However, as we already argued in Section 2, it is very common that objects are structs, with some regularity in the structure, especially in terms of first level attributes.

4.2. The SOS interface

Now that we have described the structure used for representing objects into the various data stores, we can go back to the definition of the interface illustrating the operations that can be handled. SOS provides operations that follow a general signature based on a hierarchical *path*:

- `get`(*Path*) which returns a set of objects;
- `put`(*Path*,*Object*) which inserts or updates objects;
- `delete`(*Path*) which deletes object or object fields.

The SOS interface currently exposes only one operation, PUT, to be used both for the creation and the update of objects. This is indeed the approach followed by most NoSQL systems.

The platform interprets the path and accesses objects saved either according to our common data model or using native interfaces. In detail, the query path supported by SOS in the current implementation allows the navigation of data according to the tree structure characterizing application objects. The path has the form

$step_1/step_2/\ldots/step_n$

where each *step* represents a level in the tree structure of an object (possibly indicating the collection it belongs to). Specifically, every step represents a collection, or an object identifier, or a field name. According to the most common storage practices, the first step in the path is the name of the collection where it is expected that the object is saved in. The collection name can be omitted and in such case the platform evaluates the query pattern on the whole database. Moreover the path can be used for selecting objects on the basis of the existence of some attributes. In a schemaless scenario where objects can have arbitrarily defined structures such a selection can often be enough expressive in practice. It is worth observing that the selection of objects belonging to a tree pattern is fully supported by all the storage systems handled by SOS and can be easily mapped into their structures. Let us see a few examples of paths within GET operations. The first one retrieves all the objects from the collection `users`:

```
get("users")
```

Another common case is to select a field (`name`) from an object (with identifier `281283`) in a collection (`users`):

```
get("users/281283/name")
```

The specification of a set of subobjects (a subtree in the hierarchical structure) is similar. So, the following selects all the tweets associated with the user identified by `281283`:

```
get("users/281283/tweets")
```

For storing and deleting objects, SOS provides PUT and DELETE operations, which also make use of paths, with the same logic as in the GET: every step represents a different level into the object structure where to perform the requested operation.

Let us illustrate again the major cases by means of some examples. The first one inserts (or updates, if it already exists) a field (`username`) for an object (the one with identifier `281283` in the `users` collection):

```
put("users/281283/username","mike23")
```

The second is slightly more complex, as it updates an object (with identifier `2183`) within a set (`tweets`) which is in turn a field of another object (the element of the `users` collection with identifier `281283`):

```
put("users/281283/tweets/2183", tweet)
```

Let us observe that `tweet`, the second argument of the call, is an object, whose structure is transparent.

In the DELETE operation, things are similar, so the next two operations delete the `username` field of the user whose identifier is `281283` and the tweet identified by `3463` in the collection *tweets* of user `281283`, respectively:

```
delete("users/281283/username")
delete("users/281283/tweets/3463")
```

Given the fact that the SOS common data model makes explicit reference to collections and that we assume that they have a major role in applications, we have in the

Please cite this article as: P. Atzeni, et al., Uniform access to NoSQL systems, Information Systems (2013), http://dx.doi.org/10.1016/j.is.2013.05.002

interface simplified versions of the operations, which require two simple parameters for specifying the components of interest: the name of a collection and an object identifier. Given a collection *coll* and an identifier *id*, this allows to write them as arguments, instead of using the path *coll*/*id*. Similarly, as objects very often have a struct as the first level, we have versions with three parameters, collection, identifier, and field. So, the interface exposes the following methods, together with those listed at the beginning of this section:

- `get`(*Collection*, *ObjectID*) to retrieve objects;
- `get`(*Collection*, *ObjectID*, *FieldName*) to retrieve object fields;
- `put`(*Collection*, *ObjectID*, *Object*) to insert or update objects;
- `put`(*Collection*, *ObjectID*, *FieldName*, *ObjectField*) to add or update objects fields;
- `delete`(*Collection*, *ObjectID*) to remove objects;
- `delete`(*Collection*, *ObjectID*, *FieldName*) to remove object fields.

5. Translations

In this section we illustrate how our common data model is translated into the specific structures of the various data stores and how the methods exposed by our interface can be mapped accordingly. It is worth noting that model translations and operations are two independent but related features. It is often the case that mappings between constructs cannot be designed without taking into account (or at least foreseeing) the native access patterns of the (model-specific) destination constructs.

Specifically, we describe dedicated translations for each system we provide support for. Translations are designed to be evaluated against a number of indicators such as semantic integrity, performances in query evaluation, partition friendliness. In this sense, the ultimate goal of each translation is to store data in a way that conforms to the modelling best practices of the actual data store. This way, users are not locked into accessing data from our tool, but can also effectively exploit the native interfaces of the actual systems for those specific features that are outside of the scope of SOS.

Moreover, we define alternative translations for each system. Different translations may be driven by different ways of managing semi-structured data, different access patterns, or simply by the fact that the flexibility offered by most NoSQL systems allows for many ways of exploiting the same data structures.

In the remainder of this section we show how these translations work, moving from the model generic representations to the system-specific ones. Also, translation examples will refer to the use case shown in Section 3 (a collection of Twitter users, each with personal information and a collection of tweets), a possible instance of which is the following:

```
users: [
  1001: {
    username: "bob1987",
    password: "thisisapassword",
    personal: {
      firstName: "Bob",
      lastName: "Smith",
      country: "Italy",
      city: "Rome"
    },
    tweets: [
      {
        id: "2039485563",
        text: "Hello World",
        geo: {
          longitude: "12.44751",
          latitude: "41.83764"
        }
      }
      {
        id: "4059382747",
        ...
      }
      ...
    ]
  },
  2004 {
    username: "alice",
    ...
  }
]
```

For each of the systems, we first discuss in general terms the correspondence between the common data model and the modelling features that are specific to such a system, and then illustrate how we actually implement translations with reference to the common data model in our current prototype.

5.1. HBase

Let us see how the constructs belonging to our common data model can be mapped into HBase data model. First of all, it is pretty natural to model collections as tables, as they are used to handle sets of objects. Similarly, the elements of these collections are represented by rows in the respective tables. Then, some care is needed to deal with the actual structure objects have, since our common data model allows for nesting, whereas HBase tables do not handle it directly. However, there are common practices in HBase for handling nested objects, and we will show some specific techniques based on them for representing the inner levels in nested objects.

We briefly describe the main techniques and then illustrate them by means of a couple of examples. The top level object can be a struct or a set or a simple attribute. Indeed the first case (struct) is the most common case in applications, and the other two can be dealt with by using variations of the techniques used for the first. So, let us consider the top level of an object to be a struct: clearly, its components (whichever be their type) can be stored in columns. Here we have two issues: first, how to arrange columns into families, and second, how to handle the lower levels. With respect to the first issue, there are indeed two alternatives, depending on whether objects can be assumed to have a rather regular structure (with few top level components, common to all objects, also accessed in a partitioned way) or a highly unpredictable one, with many different (potential) components or

Please cite this article as: P. Atzeni, et al., Uniform access to NoSQL systems, Information Systems (2013), http://dx.doi.org/10.1016/j.is.2013.05.002

Users			
	_top	Personal	Tweets[]
1001	username = “bob1987” password = “thisisapassword”	firstName = “Bob” lastName = “Smith” country = “Italy” city = “Rome”	[0]:id = “2039485563” [0]:text = “Hello World!” [0]:geo:longitude = “12.44751” [0]:geo:latitude = “41.83764” [1]:id = “4059382747” ...
1002	...	...	...

Usernames	
	_top
bob1987	_value = “1001”
alice	_value = “2004”
...	...

Log	
	_array[]
c1	[0] = “new user created” [1] = “query Q1 successfully applied” ...
c2	...

Fig. 4. HBase: the translation strategy with several column families.

frequent new ones for new objects. In the former case, it makes sense to use various (but few) column families, because HBase would store them separately, thus favouring accesses that involve only one or few families at the time. In the latter case the use of various column families would not help and so we use just one of them for all columns.

For what concerns the lower levels in nested objects (in both alternatives), the technique we use is based on the technique of descriptive keys we saw for Redis in Section 2: the idea is to encode a description of the structure in a column name. We demonstrate it below in the examples.

Finally, if the top level is a set or a simple attribute, then we also use a column to store it, possibly within a special (reserved) column family: in the case of the simple attribute, there would be a single value, in the case of a set there would be several.

Let us now illustrate these techniques by means of our example. Fig. 4 shows the first alternative. Table `Users` has three column families: `_top`, which is the reserved family used for simple fields, `Personal`, for storing personal data, and `Tweets[]`, for the set of tweets. Both tweets and personal information appear in most objects, so it is reasonable to dedicate separate column families for them. Let us observe that column families that model sets (`Tweets`[] in the example) are given the `[]` suffix.

These conventions are seamlessly managed by SOS, and they follow well-known modelling practices used by HBase developers. The example also shows that, if there are deeper levels in the object tree, data from each field are kept within the column family corresponding to its top level parent. For example, the `geo:latitude` and `geo:longitude` fields appear as single columns within the `Tweets[]` family, instead of generating, for instance, a dedicated `geo` family. In summary, each atomic element is stored separately, and its column contains the whole path in the tree from the family to the field itself. This strategy is effective from many points of view: it allows us to limit the number of column families to be defined in each table (which would otherwise grow, indefinitely, also at run time), and keeps (under the hypotheses for this alternative) the number of accesses needed for retrieving an object low (both for the whole object and for single parts of it).

Users	
	_uniqueFamily
1001	username = ”bob1987” password = ”thisisapassword” personal:firstName = ”Bob” personal:lastName = ”Smith” personal:country = ”Italy” personal:city = ”Rome” tweets[]:[0]:id = “2039485563” tweets[]:[0]:text = “Hello World!” tweets[]:[0]:geo.longitude = “12.44751” tweets[]:[0]:geo.latitude = “41.83764” tweets[]:[1]:id = “4059382747” ...
1002	...

Fig. 5. HBase: the translation strategy with a single column family.

The example also shows cases where the top level is not a struct. In table `Usernames`, where usernames are mapped to the correspondent ids, we have objects that are all composed of a simple value (the id itself), for which we use the `_top` column family and the `_value` qualifier. In table `Log` we store a set (indeed, a list) of log entries for each connection. These are objects composed of sets of simple objects: we use a column family named `array[]`

Fig. 5 shows the `Users` table according to the second translation strategy. Here we have just one column family: `_uniqueFamily`. The elements belonging to the set `Tweets[]` are stored with a prefix that includes the set name, thus generating qualifier names such as `tweets[]:[0]:id`.

5.2. MongoDB

As shown in Section 2, a MongoDB instance is composed of a set of named collections of BSON documents, nested up to an unbounded depth.

Therefore, there is a close correspondence with our common data model, which assumes that a database instance contains collections of hierarchical objects. MongoDB collections correspond to those of our common data model, and then the internal structure of MongoDB documents is indeed the same as that of our objects, where our struct, set, and attribute correspond to BSON object, array, and value, respectively.

Please cite this article as: P. Atzeni, et al., Uniform access to NoSQL systems, Information Systems (2013), http://dx.doi.org/10.1016/j.is.2013.05.002

As a consequence, a database instance in the common data model can be straightforwardly represented by means of a set of MongoDB collections, with a direct representation for each of them. Clearly, in this case it would not be much meaningful to look for alternate representations.

5.3. Redis

As we saw in Section 2, Redis model is quite unique within the NoSQL space, with many first level structures but no nesting. This means that we have various alternatives at the first level, but we need to resort to suitable techniques as we go deeper.

Also, our common data model has collections, whereas Redis has just one space for all its objects, identified by keys. This latter limitation can be easily overcome, by encoding in the object key also the name of the collection, following the common practice in Redis we already mentioned in Section 2. Indeed, in the same way as we saw for HBase, Redis developers follow conventions for key names that give some structure to an otherwise flat key space. Then, in the same way as we used rows for HBase, we can use hashes here for Redis, again encoding deep structures in hash fields. Given the nature of our common data model, its implementation needs only hashes and sets, the latter as support structures, as we see shortly.

The current implementation of SOS provides two different translation from the SOS common data model to the Redis data model.

The first translation (see an example in Fig. 6) adopts a simple data organization. For each object belonging to each collection one hash is created storing all the data. This is essentially the same we had in HBase, where there is a row for each object. The hash object has a key that is composed of the name of the collection and of the key of the object in the common data model (in the example, the key is `user:1001`, referring to the `user` collection and to the object with key `1001`). For each simple value, we have a field-value[7] pair, with the field name composed of a sequence that includes all the names in the path from the root to the field name, adding the suffix `[]` to the names that correspond to sets. In the same way as we did for HBase, we use again numbers in square brackets to distinguish the various elements in a set. In order to represent collections, and to be able, for example, to access all their elements, we also define one set for each collection containing all the keys of the objects of the collection. In the figure, we have the object with key `users`, which is indeed a set.

The second translation we implemented for Redis (see Fig. 7) follows a "vertical partitioning" strategy (similar to the first alternative used for HBase). This structure separates data belonging to different complex fields, following database modelling guidelines that suggest to separate objects representing different concepts. Indeed, nested

```
object key:      users
type:            set
values:          [1001, 1002, ...]

object key:      user:1001
type:            hash
values:
   username = "bob1987",
   password = "thisisapassword",
   personal:firstName = "Bob",
   personal:lastName = "Smith",
   personal:country = "Italy",
   personal:city = "Rome",
   tweets[]:[0]:id = "2039485563"
   tweets[]:[0]:text = "Hello World!"
   tweets[]:[0]:geo.longitude = "12.44751"
   tweets[]:[0]:geo.latitude = "41.83764"
   tweets[]:[1]:id = "4059382747"
   ...
```

Fig. 6. Redis: the translation strategy with one hash for every object.

data stored according to this structure can be accessed by intuitive query patterns.

We have again a set for each collection, each containing all the keys of the objects belonging to the collection itself. Then, individual objects are split into several hashes: for each complex field (set or struct) in the top level we have a specific hash, while simple fields (attributes) are stored all together in a dedicated hash, named `_top`.

The keys of the various objects are again composed of various parts, and here, since we have various hashes for each object, we have a more complex structure for keys, as in the following examples:

```
user:1001:_top
user:1001:personal
user:1001:tweets[]
```

Moreover, to keep track of the components of an object, we have an additional set, with a key that identifies the object itself (in the example, `user:1001`), containing the (suffixes of the) names of the hashes the object is made of.

As it turns out, assumptions we made for defining Redis translation are somehow close to HBase one. In fact, in our translation, Redis hashes roughly correspond to HBase column families (each containing the qualifiers map). However, object data in Redis are spread throughout many keys, whereas in HBase it is contained in a single record. As a consequence, we had to define in Redis support structures to keep track of the keys associated with each object, such as the set containing the hash names.

6. Queries

In this section we illustrate how SOS handles the specific structures of the various data stores under consideration when a GET, PUT, or DELETE operation is submitted to the interface.

6.1. HBase

Let us first consider the simplified operations illustrated at the end of Section 4.2, where collection,

[7] In hash terminology, we should say key-value, but we prefer to avoid this term to avoid confusion between this key and the key of the object.

```
object key:    users
type:          set
values:        [1001, 1002, ...]

object key:    user:1001
type:          set
values:        [_top, personal, tweets[], ]

object key:    user:1001:_top
type:          hash
values:
    username = "bob1987",
    password = "thisisapassword"

object key:    user:1001:personal
type:          hash
values:
    firstName = "Bob",
    lastName = "Smith",
    country = "Italy",
    city = "Rome"

object key:    user:1001:tweets[]
type:          hash
values:
    [0]:id = "2039485563"
    [0]:text = "Hello World!"
    [0]:geo.longitude = "12.44751"
    [0]:geo.latitude = "41.83764"
    [1]:id = "4059382747"
    ...
```

Fig. 7. Redis: the translation strategy with several hashes for each object.

identifier and possibly field are mentioned separately. Let us also recall that, according to the translations illustrated in Section 5, objects stored into HBase by our interface can be organized following two different translation patterns: the first one uses several column families (one for every first level component of the object), and the second one uses only one column family.

In this case, the interface simply identifies the table having the same name as the specified collection and performs the requested action on the row having the given object ID as key: if the action corresponds to a PUT then a new row is created or the existing one is updated; if the action is a DELETE, then the row is removed from the table, finally if the action is a GET then the object is retrieved according to the data stored in the column families of that row. For these simplified operations, the implementation of the operation is the same for the two translation policies, since object keys in both cases uniquely identify rows. The only difference is in how the data are stored and retrieved into and from the column families of the table: under the first translation pattern, SOS accesses data belonging to all the column families, whereas under the second one SOS accesses only the `_uniqueFamily` family.

In the general case of operations that specify complex paths, the implementation needs to navigate the tree structure of the object and so SOS has to map the steps of the path to the corresponding HBase structures coherently with the translation policy used for storing that data. If the first translation policy is used then the accessing pattern is interpreted as

table/*row*/*columnFamily*/*qualifier*

where the first step is mapped into a HBase table name, the second step into a row identifier, the third one into a column family and the rest into qualifiers (linking the step values for obtaining the qualifier name). Given the example described in Fig. 4, if SOS receives the following query:

```
get("users/281283/contactInfo/office")
```

then it maps users into the table `users`, contactInfo into the columnFamily `contactInfo[]`, office into the column `office`. If data are stored according to the second translation policy the accessing pattern becomes

table/*row*/*qualifier*

where the first step is mapped into a HBase table name, the second step into a row identifier, and the subsequent ones are linked together to form a qualifier name. The platform is responsible to handle the query pattern and considering `_uniqueFamily` as the default column family containing data. According to this second data organization the query:

```
get("users/281283/contactInfo")
```

maps users into the table `users` and contact data into the column `contactInfo[]` of the column family `_uniqueFamily`.

6.2. MongoDB

Given MongoDB hierarchical model and the paths that characterize document keys belonging to the path are interpreted in the following way:

collection/*id*/*field*/*subfield*/*sub*−*subfield*/…

the first element of the path corresponds to a collection name, the second one to a document ID and the following keys to nested fields. Given the correspondence between the SOS model and MongoDB one, this path translation holds both for SOS-defined databases and for those defined by the native MongoDB interface. In this way, SOS can seamlessly exploit data defined by the native interface, while, at the same time, data inserted by SOS can be exploited by the native interface as well.

In a real usage scenario, a developer would take advantage of SOS for common use cases like simple insertions and retrievals, while relying on the native libraries for complex functions that are specific to MongoDB.

6.3. Redis

When it comes to queries and path interpretation, Redis specificity requires us to adopt a slightly different approach with respect to the other data stores we support.

Given the lack of a hierarchical data model (unlike document stores with collections and objects, and extensible record stores with tables and rows), Redis developers have to store structured data making use of conventions and leveraging the different data types Redis itself provides.

Please cite this article as: P. Atzeni, et al., Uniform access to NoSQL systems, Information Systems (2013), http://dx.doi.org/10.1016/j.is.2013.05.002

As a consequence, given a hierarchical path in our interface, SOS has to figure out how the path is indeed implemented in the underlying database, sorting out a few alternative ways, and eventually returning matching results.

It is worth noting that this effort is only required for accessing data inserted by the native Redis interface, whereas SOS data are obviously guaranteed to respect our translation pattern and therefore easily retrievable.

Alternative storage patterns considered by our interface are related to the use of hashes and descriptive keys. As we discussed in Section 2, in key-value stores, hierarchical structures can be simulated by using long, composite keys whose structure describes which part of the tree the value belongs to. We also saw that hashes can be used in Redis to represent lower level structs.

As it turns out, hashes and descriptive keys can be combined in many alternative ways. In our default translation, an object is made of a number of keys, whose values are the hashes containing the actual values of the fields. Hash keys are indeed descriptive, since they are made of a sequence made of

$$collection{-}name : id : hash{-}name$$

As shown also in the translation examples, hash fields may have, in turn, descriptive keys, needed when the object hierarchy is deeper than two levels. The actual SOS key structure is therefore

$$collection/id/hash{-}name/field/subfield/sub{-}subfield/\ldots$$

This way, a path made of N steps is mapped into a descriptive key made of three steps (where the third step is the hash name), followed by a hash field with a descriptive key made of $N-3$ steps. This specific partitioning schema is reasonable but indeed quite arbitrary, making it entirely possible to find databases where, for instance, there are two levels of collections, or hashes are not used at all.

However, a general pattern can be found, by saying that a path composed of N steps can be matched by any structure identified by a descriptive key made of J steps (where the J-th step is the hash name), followed by a hash field identified by a descriptive key made of K steps, where $J+K=N$.

```
key:key:key:hash-key {
  field:field:field = value
}
```

In the example above, $J=4$ and $K=3$.

Following this pattern, SOS can be configured to access data by expecting a particular combination of J and K. Otherwise, if no instructions are given, SOS searches for matching structures for any combination of J and K. As it turns out, for paths of length N, there are exactly N possible combinations of J and K to be expected.

For instance, given any path made of 4 steps, the following structures would match:

1. `key:key:key:key = value`
2. `key:key:hash-name {`
   `field = value`
   `}`
3. `key:hash-name {`
   `field:field = value`
   `}`
4. `hash-name {`
   `field:field:field = value`
   `}`

7. The Platform

The architecture of the SOS system is organized in two main modules (Fig. 8):

- Common Interface, responsible for the methods offered by our interface.

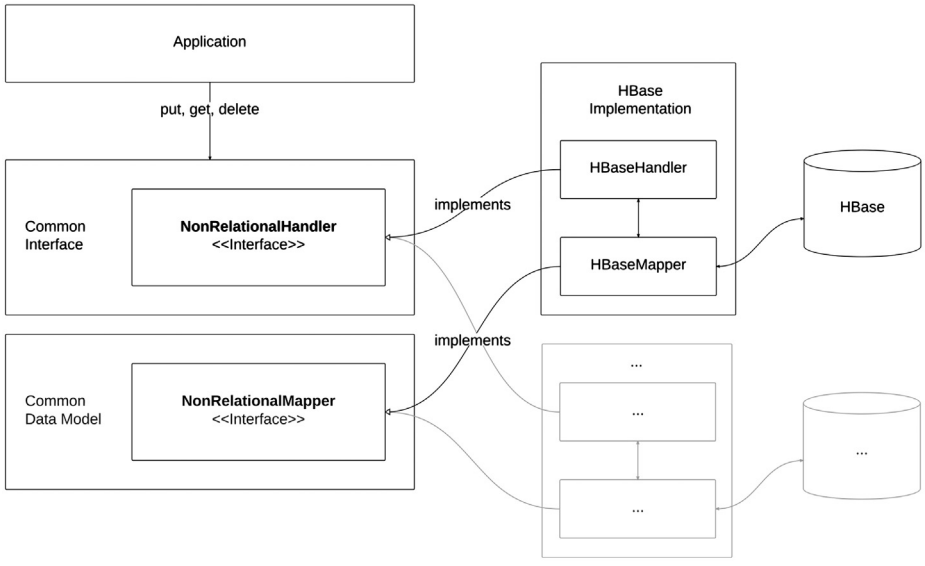

Fig. 8. Architecture of SOS.

Please cite this article as: P. Atzeni, et al., Uniform access to NoSQL systems, Information Systems (2013), http://dx.doi.org/10.1016/j.is.2013.05.002

- Common Data Model, responsible for managing the translations from our data model to the specific structures of the systems.

Each of these modules exposes a general interface: `NonRelationalHandler` for the common interface, and `NonRelationalMapper` for the common data model. The `NonRelationalHandler` is responsible for exposing `put`, `get` and `delete` operations, and for general aspects like caching and connection pooling. The main operations are defined according to the following signatures:

- `void put (String path, Object o)`
- `void delete (String path)`
- `Set < Object > get (String path)`

They are then delegated to `NonRelationalMapper`, which is responsible for the translation logic and for the access to the database. Then, `NonRelationalMapper` serializes the objects according to the tree structure of the common data model and deserializes them when a `get` request is submitted to the system.

In the current version of the tool, we implemented the tree structure in JSON, as there are many off-the-shelf libraries for Java object serialization into JSON. The implementation is based on the following mapping between the common data model and JSON format:

- Sets are implemented by arrays.
- Structs by objects.
- Attributes by values.

As the final step, each request is encoded in terms of native NoSQL DBMS operations, and the JSON object is given a suitable, structured representation, specific for the DBMS used.

We have implementations for these interfaces in the three systems we currently support. For instance, the following code[8] is the implementation of the `NonRelationalHandler` interface for MongoDB. The adapter wraps the conversion of the path into MongoDB data structure, and is responsible for calling the specific operation to a technical format (this responsibility is delegated to `objectMapper`, a shared object that all the handlers invoke in order to perform this task) which is finally persisted in MongoDB.

```
public class MongoDBHandler
    implements NonRelationalHandler {
  public void put(String path, Object obj) {
  ByteArrayOutputStream baos=
    new ByteArrayOutputStream();
  this.objectMapper.writeValue(baos, obj);
  ByteArrayInputStream bais=
    new ByteArrayInputStream(
       baos.toByteArray());
this.mongoDbMapper.persist(path,
     new ByteArrayInputStream(bais));
  baos.close();
  bais.close();
}
```

Here, method `objectMapper.writeValue(...)` serializes object *obj* according to our data model and method `mongoDbMapper.persist()` is responsible for interpreting the path according to our model and saving the object accordingly.

As a second case, let us consider the implementation of `NonRelationalHandler` for Redis. The handler at first establishes the connection with the data store (by using Jedis, a client for Redis) then, as for MongoDB, instantiates the specific mapping of Java objects into Redis manageable resources. In particular, Redis needs the concept of collection, defining a sort of hierarchy of resources, typical in resource-style architectures. It can be seen that the hierarchy is simply inferred by the mapper from the path coming from the uniform interface.

```
public class RedisNonRelationalHandler
       implements NonRelationalHandler {
  public void put(String path, Object object) {
    Jedis jedis=pool.getResource();
      ByteArrayOutputStream baos=
         new ByteArrayOutputStream();
      this.objectMapper.writeValue(baos, object);
      ByteArrayInputStream bais=
         new ByteArrayInputStream(
           baos.toByteArray());
      this.redisMapper.persist(path, bais);
      baos.close();
      bais.close();
}
```

8. Application example

In this Section we present the actual implementation of the Twitter example mentioned in Section 4.

The application is implemented by means of a small number of classes, one for users, with a method for registering new ones and for logging in, one for tweets with methods for sending them, and finally one for the "follower-followed" relationship, for updating it and for the support to listening. Each of the classes is implemented by using one or more database objects, which are instantiated according to the implementation that is desired for it (MongoDB for users, Redis for tweets, and HBase for the fellowships). More precisely, the database objects are indeed handled by a support class that offers them to all the other classes.

As an example, let us see the code for the main method, `sendTweet()` for the class that handles tweets. We show the two database objects of interest, `tweetsDB` and `followshipsDB` of the `NonRelationalHandler` with the respective constructors, used for the storage of the tweets and of the relationships, respectively. Then, the operations that involve the tweets are specified in a very simple way, in terms of `put` and `get` operations on the "DB" objects.

```
NonRelationalHandler tweetsDB=
   new RedisDbNonRelationalHandler();
NonRelationalHandler followshipsDB=
```

[8] For the sake of the readability we report here a compact version of the code that omits some details, such as initializations and exception handling.

```
      new HBaseNonRelationalHandler();
...
public void sendTweet(Tweet tweet) {
  // ADD TWEET TO THE SET OF ALL TWEETS
  tweetsDB.put("tweets/" + tweet.getId(),
       tweet);
    // ADD TWEET TO TWEETS SENT BY USER 281283
  Set <Long> sentTweets=
    tweetsDB.get("sentTweets/" + user.getId());
  sentTweets.add(tweet.getId());
  tweetsDB.put("sentTweets/" + user.getId(),
       sentTweets);
  // NOTIFY FOLLOWERS
  Set <Long> followers=
    followshipsDB.get("followers/" +
       user.getId());
  for(Long followerId : followers) {
    Set <Long> unreadTweets=
      tweetsDB.get("unreadTweets/" +
        followerId);
    unreadTweets.add(tweet.getId());
    tweetsDB.put("unreadTweets/" + followerId,
       unreadTweets);
  }
}
```

It is worth noting that the above code refers to the specific systems only in the initialization of the objects `tweetsDB` and `followshipsDB`. Thus, it would be possible to replace an underlying system with another by simply changing the constructor for these objects.

In a technical context, it is clear that an application such as the one described above can be easily implemented from scratch, given the managers for the various systems. It is important to notice that systems built on this programming model address modularity, in the sense that the NoSQL data stores can be easily replaced without affecting the client code.

9. Experiments

In this section we show how SOS main operations perform, comparing them with their native counterparts in the libraries of our supported systems.

Being performances one of the major driving factors towards the adoption of NoSQL systems, experiments will focus on evaluating the latency overhead introduced by the SOS layer in the execution of common insert/retrieve operations. However, whenever the correspondence between an SOS operation and the native one is not trivial, other aspects will be considered as well, like verbosity of the code and coupling with the specific objects structure.

9.1. Goals

Producing objective measurements about data store performances is a hard task by itself. It requires trying out different workloads, hardware configurations, and tuning clustering aspects like consistency and availability to ensure fair comparisons are produced [15]. The experiments we present here do not have the claim to provide insights about specific system behaviour and should not be interpreted as a benchmark for comparing their pure performances. They instead concentrate on evaluating SOS efficiency against that of the native systems SOS itself is built upon, within the same hardware configurations and workloads, in order to produce unbiased, repeatable results about SOS behaviour.

Also, being the SOS overhead essentially related to the back and forth translation of the objects between their Java representation, our common data model, and the data store structures themselves, we expect it to be orthogonal to the complexity of the queries executed, and to stay almost constant throughout the different operations tested on the various systems.

9.2. Hardware and software configuration

All experiments were run on a single machine configuration. However, since SOS translations run locally and in main memory, we would expect the same results also with more complex, cluster configurations. The machine has an Intel Core i7 quad-core processor running at 2 GHz, 8 GB of main memory, and a 240 GB SSD hard drive. The operative system is OSX 10.8.1.

We used MongoDB version 1.8.5, accessed through the official Java driver, version 2.5.3. Redis version is 2.4.6, accessed through the Jedis driver, version 2.0.0. HBase version is 0.90.0. Finally, Java version is 1.6.0.

9.3. Operations

Experiments focused on two main operations:

- Insertion of a single object.
- Retrieval of a single object by means of a lookup on its identifier.

The two operations correspond to the primary use of the SOS `put` and `get`, where objects are addressed by specifying their own identifier and the collection they belong to.

For both operations, we compare the SOS implementation with a custom implementation we developed out of the native interfaces the systems ship with. The custom implementation should match the correspondent SOS operation semantics, sometimes resulting in a combination of atomic native operations being used. For instance, in a PUT operation, systems are not provided with any assumption about whether the object they are going to insert may exist in advance or not. In SOS, the semantics of the PUT operation, in case of collisions on the identifier, imply the overwriting of the existing object with the new one. Actually, this is also the default behaviour of many NoSQL systems. However, for systems that behave differently (e.g. HBase), the native insertion is adapted to match SOS semantics, by deleting preemptively the identified object before the PUT itself.

9.4. Experimental setting

We developed four implementations of our use case: one using the SOS interface and three using the native interfaces of the data stores we consider (one implementation for each

Please cite this article as: P. Atzeni, et al., Uniform access to NoSQL systems, Information Systems (2013), http://dx.doi.org/10.1016/j.is.2013.05.002

data store). The various implementations handle the same data and, in order to have a meaningful comparison, native implementations have been developed by using only the basic operations that SOS provides support for. Actually, one would argue that pure native implementations could take advantage of the specific operations that data stores' interfaces provide, which would arguably provide more optimized results. However, since SOS data mapping is compatible with native interfaces one, the use of SOS does not rule out the possibility of exploiting the native operations themselves. In fact, SOS can be used together with native interfaces, thus not preventing specific optimizations on the data store. For this reason, in our experiments we only consider and compare performances of those native operations that have a counterpart in SOS, being the only ones that, in a real usage scenario, would be replaced by SOS.

In running the experiments, we considered a number of factors that could affect the objectiveness of the results. These factors are usually related to the specific behaviours of the various data stores. Some examples are:

- Caching.
- Connection pooling.
- Ramp-up period before achieving full speed.
- Impromptu events (index updating, flushing, etc.).

We also noticed that most of these factors tend to stabilize, or reach a predictable behaviour, after a number of consecutive calls (insertions/retrievals) to the data store. This threshold varies with the specific system, and, in general, is around 10,000 calls.

That said, operations are tested under three different workloads (sessions): 2000, 20,000 and 200,000 sequential calls.

Each session starts with 20,000 pilot calls, used to stabilize the data store behaviour. Then, the actual calls, used for measurements, are segmented into runs of 1000 consecutive calls. SOS runs and native runs are executed on the same database and are interlaced, to further reduce the impact of impromptu events. At the end, the duration of SOS runs is summed, and an average is computed to get the duration of the single call. The same happens for the native runs. Finally, the average overhead introduced by SOS is computed by subtracting the average native call duration from the SOS one.

9.5. Results and discussion

For each system, we ran the experiments under the three different workloads. As expected, SOS overhead is not affected by query duration and stays quite constant across different queries and systems. Also, being the SOS translations run in main memory, latency produced turns out to be very low, when compared to the entire query execution time.

As shown in Figs. 9–11, latency in insertions ranges from being almost negligible in MongoDB, where data mapping is the easiest, to slightly less than 0.1 ms in Redis. Latency in retrievals is significantly lower, and stays under 0.02 ms for all the systems tested.

It is worth noting that, being MongoDB insertions asynchronous with respect to the method call return, we are only able to reliably estimate the SOS overhead,

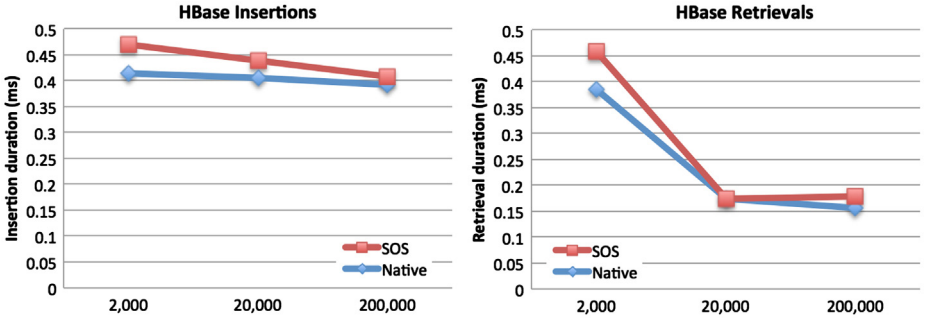

Fig. 9. HBase experiments.

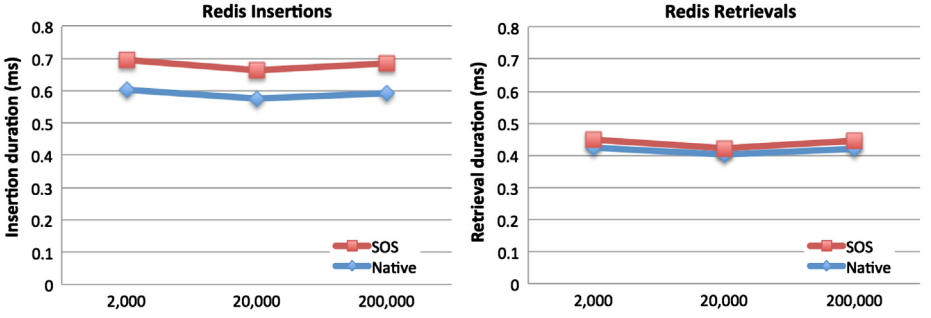

Fig. 10. Redis experiments.

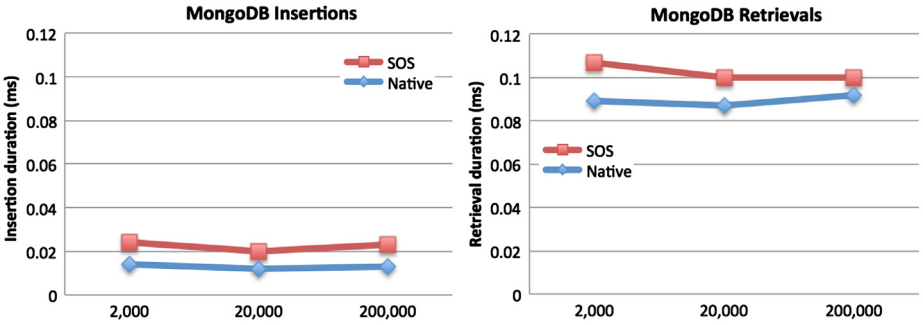

Fig. 11. MongoDB experiments.

whereas the total insertion duration should be much higher than that measured by our experiments.

As a conclusion, we believe that the overhead introduced by SOS should not be a concern in most application scenarios.

10. Related work

To the best of our knowledge, SOS is the first proposal that aims to provide support to the management of heterogeneity of NoSQL databases.

The approach for SOS we describe in this paper supports the heterogeneity of different NoSQL data models by defining a common interface that relies on a general data model. The idea of a common, pivot model finds its basis in the MIDST and MIDST-RT tools [4–6]. In MIDST, the core model (named "supermodel") is the one to which every other model converges. Whereas MIDST faces heterogeneity through explicit translations of schemas, in SOS schemas are implied and translations are not needed. The SOS common data model, therefore, is used to point out a common interface. Also, the common data model in MIDST generalizes the various models of interest, whereas in SOS it is at an intermediate level, to provide suitable support.

The need for a runtime support to interoperability of heterogeneous systems based on model and schema translation was pointed out by Bernstein and Melnik [8] and proposals in this direction, again for traditional (relational and object-oriented) models were formulated by Terwilliger et al. [21] and by Mork et al. [16]. With reference to NoSQL models, SOS is the first proposal in the runtime direction: in fact, the whole algorithm takes place at runtime and direct access to the system is granted.

From a methodological point of view, the need for a uniform classification and principle generalization for NoSQL databases is getting widely recognized; it was described by Cattell [10], reporting a detailed characterization of non-relational systems.

Stonebraker [18] presents arguments tending to diminish the importance of NoSQL systems in the scientific contest. Specifically, Stonebraker denounces the absence of a consolidated standard for NoSQL models. Also, he uses the absence of a formal query language as a supplementary argument for his thesis. Here we move from the assumption that non-relational systems have a less strict data model which cannot be subsumed under a fixed set of rules as easily as for the relational system. Indeed, our work can be a starting point for a standard interface, a common data model and a general query language.

```
User resultUser = null;
Get get = new Get(Bytes.toBytes(
      String.valueOf(281283)));
get.addFamily(Bytes.toBytes(columnFamilyName));
try {
    Result result = table.get(get);
    resultUser = new User();
    resultUser.setFirstName(Bytes.toString(
       result.getColumnLatest(
          Bytes.toBytes(columnFamilyName),
          Bytes.toBytes("firstName")).getValue()));
    resultUser.setLastName(Bytes.toString(
       result.getColumnLatest(
          Bytes.toBytes(columnFamilyName),
          Bytes.toBytes("lastName")).getValue()));
    resultUser.setId(Bytes.toLong(
       result.getColumnLatest(
          Bytes.toBytes(columnFamilyName),
          Bytes.toBytes("id")).getValue()));
    resultUser.setPassword(Bytes.toString(
       result.getColumnLatest(
          Bytes.toBytes(columnFamilyName),
          Bytes.toBytes("password")).getValue()));
    resultUser.setUsername(Bytes.toString(
       result.getColumnLatest(
          Bytes.toBytes(columnFamilyName),
          Bytes.toBytes("username")).getValue()));
} catch (IOException e) {
     e.printStackTrace();
} finally {
     return resultUser;
}
```

Fig. 12. The HBase native code for a simple SOS GET.

11. Conclusions

In this paper we introduced a programming model that enables homogeneous treatment of non-relational schemas.

We provided a common data model that allows the creation and querying of NoSQL databases defined in MongoDB, HBase and Redis using a common set of simple atomic operation. We also described an example where the interface we provide enables the simultaneous use of several NoSQL databases in a way that is transparent for the application and for the programmers.

Please cite this article as: P. Atzeni, et al., Uniform access to NoSQL systems, Information Systems (2013), http://dx.doi.org/10.1016/j.is.2013.05.002

It could be observed that such elementary operations might reduce the expressive power of the underlying databases. Actually, in this paper we do not deal with a formal analysis of information capacity of the involved models. However, it is apparent that when lower level primitives are involved, expressive power is not limited with reference to the whole language, but only to the single statement. This means that a query that can be expressed as one statement in HBase, for example, will require two or more statements in the common query language.

As an additional observation, let us comment on a couple of aspects on the code that needs to be written for applications by using SOS in comparison with the native one. The SOS code is indeed much shorter and decoupled from the actual objects structure. For example, let us consider an SOS GET:

```
databaseHandler.get("users/281283");
```

It specifies operations that would require a much longer piece of code when using native interfaces, as shown in Fig. 12 with reference to HBase. Let us also observe that the native code also needs to refer to the internal structure of the object, and therefore it would have to be rewritten if the structure evolves. Instead, the SOS code stays the same even case of changes, which is a welcome feature in a flexible world like the NoSQL field.

To the best of our knowledge this is the first attempt to reconcile NoSQL models and their programming tactics within a single framework. Our approach both offers a theoretical basis for unification and provides a concrete programming interface to address widespread problems.

We enable a sort of federated data access where different NoSQL databases can be used together in a single program. This might be even adopted in the future development of data integration tools where adapters towards NoSQL databases are still missing.

References

[1] S. Abiteboul, P. Buneman, D. Suciu, Data on the Web: From Relations to Semistructured Data and XML, Morgan Kauffman, Los Altos, 1999.

[2] S. Abiteboul, S. Cluet, T. Milo, Correspondence and translation for heterogeneous data, in: F.N. Afrati, P.G. Kolaitis (Eds.), ICDT, Lecture Notes in Computer Science, vol. 1186, Springer, 1997, pp. 351–363.

[3] S. Abiteboul, R. Hull, V. Vianu, Foundations of Databases, Addison-Wesley, 1995.

[4] P. Atzeni, L. Bellomarini, F. Bugiotti, F. Celli, G. Gianforme, A runtime approach to model-generic translation of schema and data, Information Systems 37 (3) (2012) 269–287.

[5] P. Atzeni, L. Bellomarini, F. Bugiotti, G. Gianforme, A runtime approach to model-independent schema and data translation, in: EDBT Conference, ACM, 2009, pp. 275–286.

[6] P. Atzeni, P. Cappellari, R. Torlone, P.A. Bernstein, G. Gianforme, Model-independent schema translation, VLDB Journal 17 (6) (2008) 1347–1370.

[7] P. Atzeni, V. De Antonellis, Relational Database Theory, Benjamin/Cummings, 1993.

[8] P.A. Bernstein, S. Melnik, Model management 2.0: manipulating richer mappings, in: SIGMOD Conference, ACM, 2007, pp. 1–12.

[9] M.J. Carey, N. Onose, M. Petropoulos, Data services, Communications of the ACM 55 (6) (2012) 86–97.

[10] R. Cattell, Scalable SQL and NoSQL data stores, SIGMOD Record 39 (4) (2010) 12–27.

[11] F. Chang, J. Dean, S. Ghemawat, W.C. Hsieh, D.A. Wallach, M. Burrows, T. Chandra, A. Fikes, R.E. Gruber, Bigtable: a distributed storage system for structured data, ACM Transactions on Computer Systems 26 (2) (2008).

[12] S. Cluet, C. Delobel, J. Siméon, K. Smaga, Your mediators need data conversion! in: SIGMOD Conference, 1998, pp. 177–188.

[13] E. Codd, A relational model for large shared data banks, Communications of the ACM 13 (6) (1970) 377–387.

[14] E. Codd, Relational database: a practical foundation for productivity, Communications of the ACM 25 (2) (1982) 109–117.

[15] A. Floratou, N. Teletia, D.J. DeWitt, J.M. Patel, D. Zhang, Can the elephants handle the nosql onslaught? in: PVLDB, vol. 5 (12), 2012, pp. 1712–1723.

[16] P. Mork, P. Bernstein, S. Melnik, A Schema Translator that Produces Object-to-relational Views, Technical Report MSR-TR-2007-36, Microsoft Research, 2007 ⟨http://research.microsoft.com⟩.

[17] Y. Papakonstantinou, H. Garcia-Molina, J. Widom, Object exchange across heterogeneous information sources, in: P.S. Yu, A.L.P. Chen (Eds.), ICDE, IEEE Computer Society, 1995, pp. 251–260.

[18] M. Stonebraker, Stonebraker on NoSQL and enterprises, Communications of the ACM 54 (August) (2011) 10–11.

[19] M. Stonebraker, R. Cattell, 10 rules for scalable performance in 'simple operation' datastores, Communications of the ACM 54 (6) (2011) 72–80.

[20] M. Stonebraker, S. Madden, D.J. Abadi, S. Harizopoulos, N. Hachem, P. Helland, The end of an architectural era (it's time for a complete rewrite), in: VLDB, 2007, pp. 1150–1160.

[21] J.F. Terwilliger, S. Melnik, P.A. Bernstein, Language-integrated querying of xml data in sql server, in: PVLDB, vol. 1 (2), 2008, pp. 1396–1399.

Please cite this article as: P. Atzeni, et al., Uniform access to NoSQL systems, Information Systems (2013), http://dx.doi.org/10.1016/j.is.2013.05.002