Uniform access to non-relational database systems: the SOS platform

Paolo Atzeni, Francesca Bugiotti, and Luca Rossi

Dipartimento di informatica e automazione
Università Roma Tre

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 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 (and also to relational ones) 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 metamodeling 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.

Keywords: Non-Relational databases, NoSQL, interoperability

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 [12] and provide scalability [11]. 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 [11]. 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 domains, 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 also relax consistency requirements. 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 [6, 11], more than fifty, and each of them exposes a different interface (different 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 [10]: applications and data are not portable and skills and expertise acquired on a specific system are not reusable with another one. Also, most of the interfaces support a lower level than SQL, with record-at-a-time approaches, which appear to be a step back with respect to relational systems.

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 also can enable interoperability between them together with ease of development (by improving programmers' productivity, following one of the original goals of the relational databases [8]). As a first step in this direction, we present here *SOS (Save Our Systems)*, a programming environment where non-relational databases can be uniformly defined, queried and accessed by an application program.

The programming model is based on a high-level description of the interfaces of non-relational systems by means of a generic and extensible meta-layer, based on principles that are inspired by those our group has used in the MIDST and MIDST-RT projects [2–4]. The focus in our previous work was on the structure of models and was mainly devoted to the definition of techniques to translate from a representation to another one. Here, the meta layer refers to the basic common structure and it is therefore concerned with the methods that can be used to access the systems. The meta-layer represents a theoretical unifying

structure, which is then instantiated (indeed, implemented) in the specific underlying systems; 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), and this could really be important in the tumultuous world of Internet applications. 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. Indeed, we will show a simple application which involves different systems and can be developed in a rapid way by knowing only the methods in our interface and not the details of the various underlying 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. Indeed, we are currently working at additional components that would allow code written for a given system to access other systems: this will be done by writing a layer to translate proprietary code to the SOS interface; then, the tool proposed here would allow for the execution on one system of code developed for another one.

The rest of this paper is organized as follows. In Section 2 we illustrate the approach by defining a common interface. In Section 3 we illustrate our metalayer and in Section 4 we present the structure of the platform. In Section 5 we illustrate a simple example application. Then, we briefly discuss related work (Section 6) and draw our conclusions (Section 7).

2 The common interface

As we said, the goal of our approach is to offer a uniform interface that would allow access to data stored in different data management systems (mainly of the NoSQL family, but possibly also relational), without knowing in advance the specific one, and possibly using different systems within a single application. In this section we discuss on the desirable features of such an interface and then present our proposal for it. In the next section we will then describe the underlying architecture that allows for mapping it to the various systems.

NoSQL systems have been developed independently from one another, each with specific application objectives, but all with the general goal to offer rather simple operations, to be supported in a scalable way, with possibly massive throughput.[4] Indeed, there are many proposals for them, and effort have been

[1] `http://redis.io`
[2] `http://www.mongodb.org`
[3] `http://hbase.apache.org`
[4] Our interest here is in the features related to how data is modeled and accessed. So, we will not further refer to the attention to scalability nor to another important

devoted to the classification of them into various categories, the most important of which have been called key-value stores, document stores and extensible record stores, respectively [6]. The three families share the idea of handling individual items ("objects" or "rows") and on the need for not being rigid on the structure of the items, while they differ on the features that allow to refer to internal components of the objects.[5] Most importantly, given the general goal of concentrating on simple operations, they are all based on simple operations for inserting and deleting the individual items, mainly one at the time, and retrieving them, one at the time or a set at the time. Therefore, a simple yet powerful common interface can be defined on very basic and general operations:

- **put** to insert objects
- **get** to retrieve objects
- **delete** to remove objects

Crucial issues in this interface are (i) the nature of the objects that can be handled, which in some systems are allowed to have a rather complex structure, not fixed in advance, but with components and some forms of nesting, and in others are much simpler and (ii) the expressivity of the get operation, which in some cases can only refer to identifiers of objects and in others can be based on conditions on values. This second issue is related to the fact that operations can also be specified at high level with reference to a single field of a structured object. The platform infers from the meta-layer all the involved structures and defines coherently the sequence of operations on them.

The simple interface we have just described is the core component of the architecture of the SOS system, which is organized in the following modules (Figure 1):

- the common interface that offers the primitives to interact with NoSQL stores;
- the meta-layer that stores the form of the involved data;
- the specific handlers that generate the appropriate calls to the specific database system.

The SOS interface exposes high level operations with reference to the general constructs defined in the meta-layer. The meta-layer stores data and provides the interface with a uniform data model for performing operations on objects. The specific handlers support the low level interactions with specific NoSQL storage systems mapping meta-layer generic calls to system specific queries.

In order to show how our proposed system can support application development, in this paper we will refer to an example regarding the definition of a simplified version of Twitter,[6] the popular social network application. We will

issue, the frequent relaxation of consistency, which are both orthogonal to the aspects we discuss. Because of this orthogonality, our approach preserves the benefits of scalability and relaxation of consistency.

[5] We will give some relevant details for the three families and their representative systems in the next section, while discussing how the common interface can be implemented in them.

[6] `http://twitter.com/`

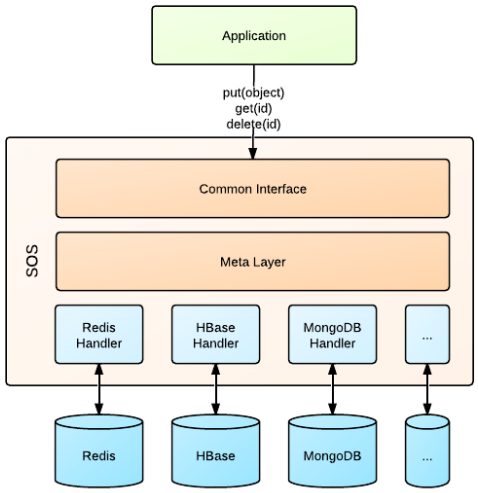

Fig. 1. Architecture of SOS

adopt the perspective of a Web 2.0 development team that wants to benefit from the use of different NoSQL systems. Transactions are short-lived and involve little amount of data, so the adoption of NoSQL systems is meaningful. Also, let us assume that quantitative application needs have led the software architect to drive the decision towards the use of several NoSQL DBMSs, as it turned out that the various components of the application can benefit each from a different system.[7]

The data of interest for the example have a rather simple structure, shown in Figure 2: we have users, with login and some personal information, who write posts; every user "follows" the posts of a set of users and can, in turn, "be followed" by another set of users. In the example, in Section 5, we will show how this can be implemented by using three different NoSQL systems in one single application.

[7] 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 behavior in terms of consistency, it is meaningful to have applications that are not satisfied with just one of them.

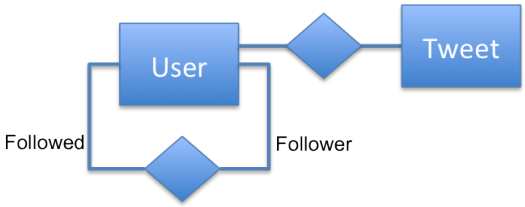

Fig. 2. The schema for the data in the example

3 The Meta-layer

In this section we give some formal explanation of the meta-layer structure and we contextualize it with respect to the NoSQL system our platform handles.

Our approach leverages on the genericity of the data model to allow for a standard development practice that is not bound to a specific DBMS API, but to a generic one.

Application programming interfaces are built over and in terms of the constructs of the meta-layer (without explicit reference to lower level constructs); in this way, programs are modular and independent of the particular data model; reuse is maximized.

According to the literature, a data model can be represented as the collection of its characterizing constructs, a set of constraints and a set of operations acting over them [14]. A construct is an entity that has a conceptual significance within the model. A construct can be imagined as the elementary outcome of a structural decomposition of a data model. A construct is relevant in a data model since it is a building block of its logical structures.

Thus, the aim of the meta-layer is to reconcile the relevant descriptive elements of mainstream NoSQL databases: key-value stores, document stores and record stores. In the following paragraphs, we will describe concisely their data models, and we will show how their constructs and operations can be effectively modeled through our meta-layer.

In key-value stores, data is organized in a map fashion: each value is identified by a unique key. Keys are used to insert, retrieve and remove single values, whereas operations spanning multiple ones are often not trivial or not supported at all. Values, associated with keys, can be simple elements such as Strings and Integers, or structured objects, depending on the expressive power of the DBMS considered. We chose Redis as a representative of key-value stores, being one of the richest in terms of data structures and operations. Redis supports various data types such as Set, List, Hash, String and Integer, and a collection of native operations to manipulate them.

Document stores handle collections of objects represented in structured 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, MongoDB is one of the most adopted, offering a rich programming interface for manipulating both entire documents and individual single fields.

Extensible record stores offer a relaxation of the relational data model, allowing tables to have a dynamic number of columns, grouped in families. Column families are used for optimization and partitioning purposes. Within a table, each row is identified by a unique key: rows are usually stored in lexicographic order, which enables single accesses and sequential scans as well. The progenitor of this category is Google BigTable [7] and we consider here HBase, which is modeled after it and supports most of the features described above.

Moving from the data models described above, our meta-layer is designed for dealing with them effectively, allowing to manage collections of objects having an arbitrarily nested structure. It turns out that this model can be founded on three main constructs: Struct, Set and Attribute.

Instances of each construct are given a name associated to a value. The structure of the value depends on the type of construct itself: Attributes contain simple values, such as Strings or Integers; Structs and Sets, instead, are complex elements whose values may contain both Attributes and Sets or Structs as well, as shown in Figure 3.

Each database instance is represented as a Set of collections. Each collection is a Set itself, containing an arbitrary number of objects. Each object is identified by a key, which is unique within the collection it belongs to.

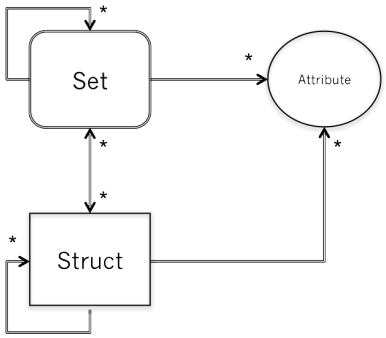

Fig. 3. The metalayer

As it turns out, specific non-relational models can be represented as a particular instance of the meta-layer, where generic constructs are used in well-defined combinations and, if necessary, renamed. Simple elements, such as key-values pairs or single qualifiers can be modeled as Attributes. Groups of attributes, as HBase column families or Redis hashes, are represented by Structs. Finally, collections of elements, as HBase tables or MongoDB collections, are modeled by

Sets. According to the specific structures the various NoSQL systems implement, we will define a translation process, from meta-layer constructs, to coherent system-specific structures. In the remainder of this section it will be shown how this translation and data memorization works, moving from the model generic representation to the system-specific ones, with reference to the example introduced in Section 2. The meta-layer representation of the example is shown in Figure 4. A Struct Tweet represents a tweet sent by a User and a Set Tweets contains all the tweets of a User. Finally a simple Attribute is used for every non-structured information item of Users and Tweets (SSN, Name, Surname, ...).

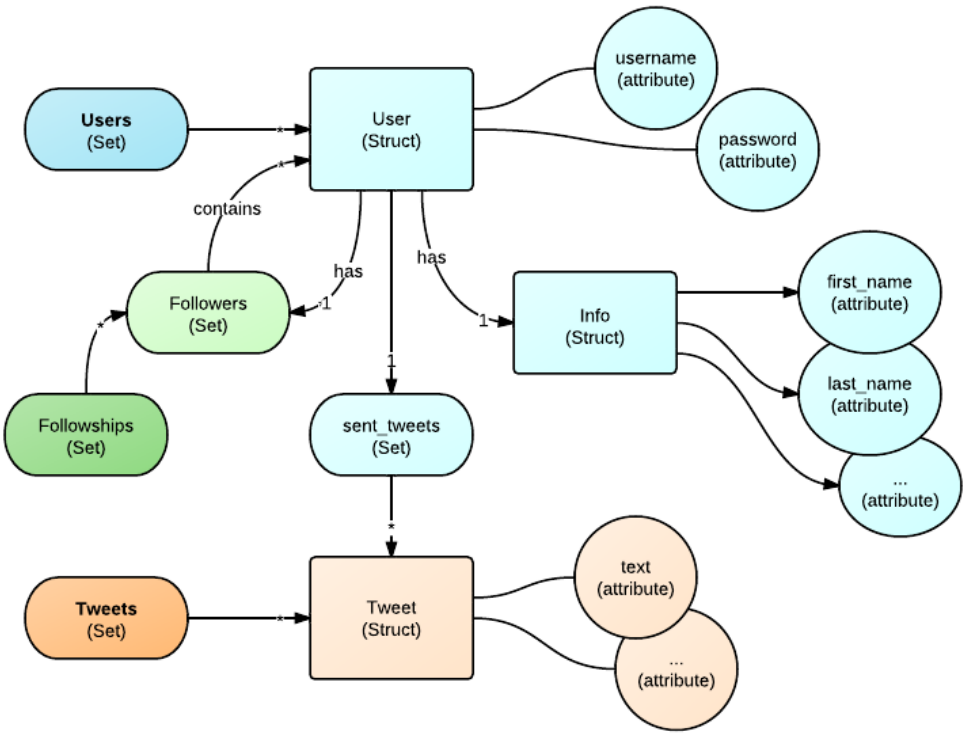

Fig. 4. The meta-layer representation

3.1 MongoDB

MongoDB handles Collections of structured documents represented in BSON[8] and identified by a global key. In our translation, every SOS Collection is im-

[8] BSON: a binary encoding of the popular JSON format

plemented as a MongoDB Collection. Each object in the meta-layer is then represented as a single document within the collection itself. The structure of each object, implemented in JSON, fits closely the BSON format, allowing seamless translations between the two models.

3.2 HBase

In this context we model every SOS Collection as a Table. Objects within a Collection are implemented as records in the corresponding Table: the structure of each object establishes which HBase constructs are involved in the translation. Top-level Attributes are stored in a reserved column family named _top within a reserved column (qualifier) named _value; top-level Sets generate a reserved column family (named _array[]) that groups all the fields they further contain. Finally, each top-level Struct is represented as a single column family; deeper elements it contains are further stored in qualifiers. In order to store a nested tree in a flat structure (i.e. the qualifiers map) each field of the tree is given a unique qualifier key made of the whole path in the tree that goes from the root Struct of the column family, to the field itself.

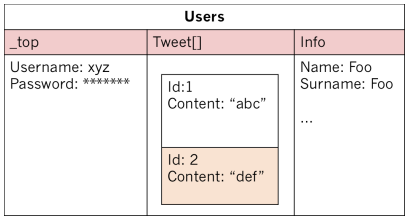

Fig. 5. HBase Example

An example of translation is shown in Figure 5. The choice of storing top-level elements into different column families is driven by some data modeling considerations. In HBase, column families correspond to the first-class concepts each record is made of; in fact, data partitioning and query optimization are tuned on a per-column-family basis, considering each column family as an independent conceptual domain. In tree documents, we assume that top-level structs and sets correspond to the most significant concepts in the model, and therefore we represent each of them as a single column family.

3.3 Redis

Redis, among the three DBMSs we consider, is arguably the most flexible and rich in constructs: it is a key-value store where values can be complex elements such as Hashes, Sets[9] or Lists.

[9] In order to avoid ambiguity, we will use the term R-Set to refer to a Redis set.

For every object of the SOS Collections of the meta-layer, in Redis two R-Sets are defined. The first one is used to store keys, therefore it indexes the elements. The second one contains data: a Hash named _top with Attributes directly related to the Struct and a different Hash for every Struct or Set it contains. The name of those Hashes will be the concatenation of the name of the elements and the contained Struct.

The top Hash will contain all the first level attributes as following:

```
hash key:          user:10001
hash values:       [_top, info, twees[], ]

hash key:          user:10001:\_top
hash values:       username: foo
                   password: bar

hash key:          user:10001:info
hash values:       firstName: "abc"
                   lastName:  "def"

hash key:          user:10001:tweets[]
hash values:       [0].tweet.id = "1234",
                   [0].tweet.content = "....",
                   [1].tweet.id = "1235"
                   [1] ...
```

Assumptions we made for defining Redis translation are somehow close to HBase ones, discussed above. In fact, in our translation, Redis Hashes roughly correspond to HBase column families (each containing the qualifiers map). Nevertheless, object data in Redis is spread throughout many keys, whereas in HBase it is contained in a single record. As a consequence, we have to define in Redis support structures to keep track of the keys associated with each object, such as the R-set containing the hash names.

4 The platform

For the definition of the SOS platform, we featured a Java API that exposes the following methods corresponding to the basic operations illustrated in Section 2:

- `void put (String collection, String ID, Object o)`
- `void delete (String collection, String ID)`
- `Object get (String collection, String ID)`
- `Set<Object> get (Query q)`

The core class is `NonRelationalManager`. It supports `put`, `delete` and `get` operations. Primitives are based on object identifiers, however multiple retrievals are also possible by means of simple conjunctive queries (the second form of `get`). These methods handle arbitrary Java objects and are responsible for their serialization into the target NoSQL system. This process is based on the meta-layer, the data model pivoting the access to the systems. In the current version

of the tool, we implemented the meta-layer 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 meta-layer 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. The requests and the interactions are handled by technology-specific implementations acting as adapters for the DBMS API.

We have implementations for this interface in the three systems we currently support. The classes that directly implement the interface are the "managers" for the various systems, which then delegate to other classes some of the technical, more elaborate operations.

For example, the following code is the implementation of the `NonRelationalManager` interface for MongoDB. It can be noticed that `put` links a content to a resource identifier, indeed creates a new resource. The adapter wraps the conversion to a technical format (this responsibility is delegated to `objectMapper`) which is finally persisted in MongoDB.

```
public class MongoDBNonRelationalManager
               implements NonRelationalManager {

   public void put(String collection, String ID, Object object) {
      DBCollection coll = db.getCollection(collection);

      ByteArrayOutputStream baos =
                    new ByteArrayOutputStream();
      this.objectMapper.writeValue(baos, object);

      this.mongoMapper.persist(coll, getId(ID),
               new ByteArrayInputStream(baos.toByteArray()));
}
```

As a second implementation of `NonRelationalManger`, let us consider the one for Redis. As for MongoDB, it contains 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 from the ID coming from the uniform interface.

```
public class RedisNonRelationalManager
                implements NonRelationalManager {

public void put(String collection, String ID, Object object) {
  Jedis jedis = pool.getResource();
```

```
  try {

     // the object is stored in the meta-layer
     ByteArrayOutputStream baos =
               new ByteArrayOutputStream();
     this.objectMapper.writeValue(baos, object);

     ByteArrayInputStream bais =
               new ByteArrayInputStream(baos.toByteArray());
     this.databaseMapper.persist(
               jedis, collection, ID, bais);

     baos.close();
     bais.close();

     } catch(JsonParseException ex) {
          ex.printStackTrace();

     } finally {
          pool.returnResource(jedis);
     }
}
```

5 Application example

In this Section we present the actual implementation of the Twitter example mentioned in Section 2.

The application can be 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 followships). 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 `NonRelationalManager` 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.

```
     NonRelationalManager tweetsDB =
           new RedisDbNonRelationalManager();
     NonRelationalManager followshipsDB =
           new HBaseNonRelationalManager();
  ...
```

```
public void sendTweet(Tweet tweet) {

   // ADD TWEET TO THE SET OF ALL TWEETS
   tweetsDB.put("tweets", tweet.getId(), tweet);

   // ADD TWEET TO THE TWEETS SENT BY THE USER
   Set<Long> sentTweets =
     tweetsDB.get("sentTweets", tweet.getAuthor());
   sentTweets.add(tweet.getId());
   tweetsDB.put("sentTweets", tweet.getAuthor(), sentTweets);

   // NOTIFY FOLLOWERS
   Set<Long> followers =
      followshipsDB.get("followers", tweet.getAuthor());

   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 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 infrastructure can be easily replaced without affecting the client code.

6 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 here uses the meta-layer as the principal means to support the heterogeneity of different data models. The idea of a pivot model finds its basis in the MIDST and MIDST-RT tools [4, 3]. 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 pivot model, the meta-layer, is used to point out a common interface. Several other differences exist between the two approaches, as we already remarked in the introduction.

The need for a runtime support to interoperability of heterogeneous systems based on model and schema translation was pointed out by Bernstein and Melnik [5] and proposals in this direction, again for traditional (relational and object-oriented) models were formulated by Terwilliger et al. [13] and by Mork et al. [9]. 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 theoretical point of view, the need for a uniform classification and principle generalization for NoSQL databases is getting widely recognized; it was described by by Cattell [6], reporting a detailed characterization of non-relational systems.

Stonebraker [10] presents a radical approach tending to diminish the importance of NoSQL systems in the scientific contest. Actually, 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. However, we notice that commonalities and structural concepts among the various systems can be detected and this can be exploited to build a common meta-layer.

7 Conclusions

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

We provided a meta-layer 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 it is transparent for the application and for the programmers.

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.

However, what is noticeable is that the standardization is not achieved by a adding a supplementary level of abstraction but by spotting common concepts in the models which are deemed general.

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 programming 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 still lack.

The solution proposed here adds an indirection layer to the generally bare-boned NoSQL databases. Obviously, this homogenization presents performance tradeoffs that need to be carefully studied.

References

1. S. Abiteboul, P. Buneman, and D. Suciu. *Data on the web: from relations to semistructured data and XML*. Morgan Kaufmann, 2000.
2. P. Atzeni, L. Bellomarini, F. Bugiotti, F. Celli, and G. Gianforme. A runtime approach to model-generic translation of schema and data. In *Inf. Syst.*, pages 269–287, 2012.
3. P. Atzeni, L. Bellomarini, F. Bugiotti, and G. Gianforme. A runtime approach to model-independent schema and data translation. In *EDBT Conference*, pages 275–286. ACM, 2009.
4. P. Atzeni, P. Cappellari, R. Torlone, P. Bernstein, and G. Gianforme. Model-independent schema translation. *VLDB Journal*, 17(6):1347–1370, 2008.
5. P. A. Bernstein and S. Melnik. Model management 2.0: manipulating richer mappings. In *SIGMOD Conference*, pages 1–12, 2007.
6. R. Cattell. Scalable SQL and NoSQL data stores. *SIGMOD Record*, pages 12–27, 2010.
7. F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26(2):4:1–4:26, 2008.
8. E. Codd. Relational database: A practical foundation for productivity. *Commun. ACM*, 25(2):109–117, 1982.
9. P. Mork, P. Bernstein, and S. Melnik. A schema translator that produces object-to-relational views. Technical Report MSR-TR-2007-36, Microsoft Research, 2007. http://research.microsoft.com.
10. M. Stonebraker. Stonebraker on NoSQL and enterprises. *Commun. ACM*, pages 10–11, 2011.
11. M. Stonebraker and R. Cattell. 10 rules for scalable performance in 'simple operation' datastores. *Commun. ACM*, pages 72–80, 2011.
12. M. Stonebraker, S. Madden, D. J. Abadi, S. Harizopoulos, N. Hachem, and P. Helland. The end of an architectural era: (it's time for a complete rewrite). In *VLDB*, pages 1150–1160, 2007.
13. J. F. Terwilliger, S. Melnik, and P. A. Bernstein. Language-integrated querying of XML data in SQL server. *PVLDB*, pages 1396–1399, 2008.
14. D. Tsichritzis and F. Lochovski. *Data Models*. Prentice-Hall, Englewood Cliffs, New Jersey, 1982.