SOS (Save Our Systems): A uniform programming interface for non-relational systems

Paolo Atzeni
Università Roma Tre
atzeni@dia.uniroma3.it

Francesca Bugiotti
Università Roma Tre
franbugiotti@yahoo.it

Luca Rossi
Università Roma Tre
luca.rossi.917@gmail.com

ABSTRACT

The recent growth of non-relational databases (often termed as NoSQL) is an interesting phenomenon that has generated both interest and criticism. One of the major drawbacks that is often referred to is the heterogeneity of the languages and interfaces they offer to developers and users.
SOS is proposed as a common interface to them, in order 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 various 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. The demonstration will focus on a simple yet powerful application scenario which accesses three different NoSQL systems.

Categories and Subject Descriptors

H.2.5 [**Heterogeneous databases**]: Data translation, Program translation

Keywords

Metamodeling, Nonrelational databases

1. INTRODUCTION

Relational data base systems (RDBMSs), as developed over the last thirty or even forty years, dominate the market by providing an integrated set of services that refer to a variety of requirements, which include support for both traditional transaction processing and decision support, as well as to complex operations of various kinds. All the major RDBMSs on the market show a similar architecture (based on the first systems developed in the Seventies) and do support SQL as a standard language (even if with dialects that differ somehow). They do provide reasonably general-purpose solutions that balance in an often satisfactory way the various requirements.

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 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 in other cases, with arguments that call for *semistructured* data (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, that of social networking, 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 for simple operations only (and so, limited support for 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. An interesting classification has been proposed, based on the modeling constructs available [6, 11]: key-value stores (representatives of which are Redis and Scalaris), document stores (including MongoDB and CouchDB) and extensible record stores (including BigTable, HBase and Cassandra). There exist more than fifty systems, in the various categories, and each of them can be used by means of a different interface (different model and different API). Indeed, as it has been recently pointed out, the lack of a standard is a great concern for any organization interested in adopting any of these systems [10]: applications 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 relational databases [8]). As a first step in this direction, we present here *SOS (Save Our Systems)*, a programming interface where non-relational databases can be uniformly defined, queried and accessed by an application program.

The tool makes use of a description of the interfaces of non-relational systems by means of a generic and extensible hierarchical meta-layer, based on principles that are inspired by those our group has used in the MIDST and MIDST-RT projects [2, 3, 4]. However, while the focus in our previous work was on the structure of models, here, as models are exposed to a limited extent, the meta-layer is concerned with the methods that can be used to access the systems. The meta-layer is then instantiated (indeed, implemented) in the various underlying systems; in this demonstration, we will use the implementations for a representative for each of the main classes we mentioned earlier: a key-value store, Redis, a document store, MongoDB, and an extensible data store, HBase. 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. 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.

The main contribution of this demonstration is in its originality, as there is no other tool 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 other target systems.

The rest of this paper is organized as follows. In Section 2 we illustrate the principles on which the SOS tool is based. In Section 3 we illustrate the application we will demonstrate. Then, we briefly discuss related work (Section 4) and draw our conclusions (Section 5).

2. THE MAIN CONCEPTS

The architecture of the tool and its interaction with applications and NoSQL systems are summarized in Figure 1. In this section we describe the main features.

As we said, NoSQL systems provide different interfaces to access the data they manage, and we have the goal of providing some support for reducing the difficulties related with this heterogeneity. Indeed, despite their diversity, these systems share some commonalities:[1] the flexibility in the data model and the low level of the operation on data.

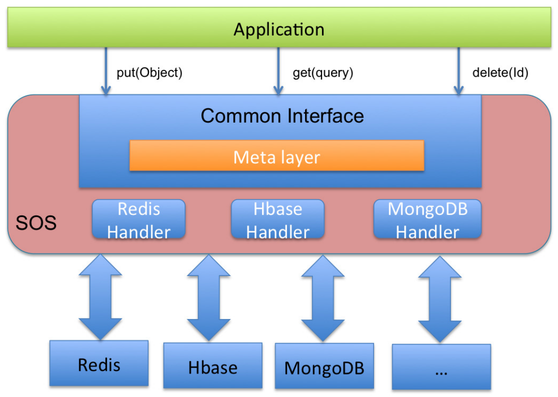

Figure 1: Architecture of SOS

More specifically, these systems all provide operations on objects, which have some form of identifier. Each object has an internal structure that has some form of components for each object, with different features in the various systems.

In our tool we currently provide support for three NoSQL systems: HBase, Redis and MongoDB, as representatives of the three major categories mentioned in the Introduction. HBase[2] is an extensible record store modeled after Google BigTable [7]. Data are organized in tables where each row has an arbitrary number of fields (columns) grouped in clusters (column families). It is possible to retrieve rows using identifiers (single values, yielding one row, or ranges, producing collections of rows). HBase does not support any notion of query to retrieve rows on conditions on fields other than the identifier. Redis[3] is a key-value store where single values can be retrieved using their identifier. Redis supports various data types such as String, Integer, Set, and List and native operations to manipulate them. Finally, MongoDB[4] is a document store that handles collections of objects represented in BSON[5] a variant of the JSON[6] format. Every document has a unique identifier that is used for indexing purpose. MongoDB couples a very simple and intuitive data organization with a rich interface for querying and manipulating data.

It turns out that these three systems (as well as most of the others in the NoSQL world) support operations that, with differences in details related to the specific data organization and the syntactic choices for the languages, provide support for storing and removing objects and for retrieving objects or sets of objects. As a consequence, in our search for a unified approach, we decided to design an interface that offers put, delete and get methods, the latter referring to both uniquely identified objects and to sets of them.

In order to handle the representation of objects that are semistructured, with various degrees of structuredness, SOS

[1]We refer here to the features related to how data is modeled and accessed. There are other common, important features, namely the attention to scalability and the frequent relaxation of consistency, which are orthogonal to the aspects we are interested in here, and so we do not consider them.

[2]`http://hbase.apache.org`

[3]`http://redis.io`

[4]`http://www.mongodb.org`

[5]`http://bsonspec.org`

[6]`http://www.json.org`

adopts an intermediate, common representation for objects, implemented in the JSON format, which offers a compact hierarchical organization which is, at the same time, flexible, as the internal structure of objects can be used or ignored, depending on the capabilities of the underlying systems. Moreover, JSON documents are easily handled by most programming languages, and often used as a suitable format for serializing their native objects.

These ideas could be implemented in various programming paradigms and environments. As a first effort, we decided to proceed with Java, given the large set of applications that use it.

So, we defined a Java interface (named `NonRelational-Handler`), which exposes the following methods corresponding to the basic operations illustrated above:

```
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)
```

So, `NonRelationalHandler` supports `put`, `delete` and `get` operations based on the objects identifiers, as well as multiple retrievals by simple conjunctive queries (the second form of `get`). These methods handle arbitrary Java objects, which are then serialized into JSON documents in order to be processed by the underlying layer. Finally, each request is codified 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' drivers.

We have implementations for this interface in the three systems we currently support. The classes that directly implement the interface are the "handlers" 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 `NonRelationalHandler` interface for MongoDb. It can be seen 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 MongoDBNonRelationalHandler
               implements NonRelationalHandler {
   public void put(String cName, String ID, Object object) {
     DBCollection coll =
         db.getCollection(this.getCollection(cName));
      ByteArrayOutputStream baos =
                    new ByteArrayOutputStream();
      this.objectMapper.writeValue(baos, object);
      ByteArrayInputStream bais =
               new ByteArrayInputStream(baos.toByteArray());
      this.mongoMapper.persist(coll, ID, bais);
}
```

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.

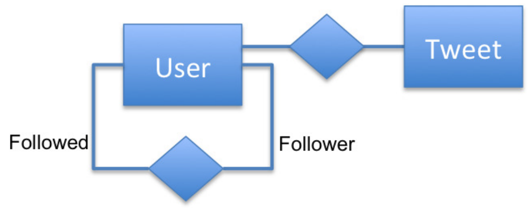

Figure 2: The schema for the data in the example

```
public class RedisNonRelationalHandler
                implements NonRelationalHandler {
public void put(String collName, 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);
     DBCollection coll =
         db.getCollection(this.getCollection(cName));
     ByteArrayInputStream bais =
               new ByteArrayInputStream(baos.toByteArray());
     this.databaseMapper.persist(
               jedis, coll, ID, bais);
     baos.close();
     bais.close();
     } catch(JsonParseException ex) {
          ex.printStackTrace();
     } finally {
          pool.returnResource(jedis);
     }
}
```

3. THE DEMO APPLICATION

In order to show how the SOS tool can support application development, let us adopt the perspective of a Web 2.0 development team in charge of releasing a new version of Twitter. 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 lead 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. We store users' data in MongoDB, posts in Redis and the "follower-followed" relationships in HBase.

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 for 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,

[7]For the sake of space here (and of time in the demo), 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.

Redis for tweets, and HBase for the relationship). 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, named `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 MongoDbNonRelationalHandler();
    NonRelationalHandler followshipsDB =
          new HBaseNonRelationalHandler();
...
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`. So, it would possible to replace an underlying system with another by simply changing the constructor for these objects.

In the demo, we will show how an application such as the one we have just described can be easily implemented from scratch, given the handlers for the various systems, and we will also show how immediate is to replace a NoSQL system with another, given our tool.

4. RELATED WORK

To the best of our knowledge SOS is the first proposal that aims to provide a solution to handle the heterogeneity of NoSQL databases. The approach for SOS we describe in this demo finds its basis in the MIDST and MIDST-RT tools [2, 3, 4], where we defined a generic metamodel for representing and querying a wide set of models for traditional databases. The SOS interface uses the same principles and ideas of MIDST metalayer, but with a number of differences, as we already pointed out in the introduction. The need for a run-time support for 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]. This is the first such a proposal in the NoSQL field.

The problem of classifying and finding common principles for NoSQL databases was described by Cattell [6] where non-relational systems are characterized in detail. Stonebraker [10] describes in detail the problems in using NoSQL databases deriving from the absence of a standard model and a common query language and interface.

5. CONCLUSIONS

In this paper we introduced a programming interface 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. Finally we described an example where, the interface we provide enables the contemporary use of NoSQL database transparently for the application and for the programmers.

6. REFERENCES

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

[2] P. Atzeni, L. Bellomarini, F. Bugiotti, F. Celli, and G. Gianforme. A runtime approach to model-generic translation of schema and data. *Inf. Syst.*, 37(3):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, ACM*, pages 275–286, 2009.

[4] P. Atzeni, P. Cappellari, R. Torlone, P. A. Bernstein, and G. Gianforme. Model-independent schema translation. *VLDB J.*, 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. ACM, 2007.

[6] R. Cattell. Scalable SQL and NoSQL data stores. *SIGMOD Record*, 39(4):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), 2008.

[8] E. Codd. Relational database: A practical foundation for productivity. *CACM*, 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*, 54:10–11, August 2011.

[11] M. Stonebraker and R. Cattell. 10 rules for scalable performance in 'simple operation' datastores. *Commun. ACM*, 54(6):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*, 1(2):1396–1399, 2008.