JSON-based interoperability applying the pull-parser programming model

Leandro Pulgatti[1] Marcos Didonet Del Fabro[1]
[1]*C3SL Labs, Federal University of Paraná, Curitiba, Brazil*
{*ldpulgatti, marcos.ddf*}*@inf.ufpr.br*

Keywords: NoSQL models · JSON interoperability · Pull-parser programming model

Abstract: The JSON format is been applied in a variety of applications: it is established as the de-facto standard for representing document stores; it is widely used to achieve interoperability and as the exchange format in RESTful web APIs. For these reasons, it is necessary to provide interoperability between JSON and other NoSQL formats. There are several approaches that aims to translate between different NoSQL formats, however, most of them attempt to be generic and do not focus on JSON. They aim on providing an abstract and generic representation capturing all the data models constructs and to provide wrapper-like structures, or to develop pairs of translators. In this paper, we present an approach that uses the JSON data model as driving format for interoperability with distinct NoSQL data models. We take advantage of its nested textual structure to apply the pull-parser programming model to process it and to develop translators between JSON and a set of representative NoSQL formats. We focus on the JSON extraction and on the development and application of the data transformations. We validate our approach through an implementation handling a large number of data representation strategies.

1 INTRODUCTION

The JSON (Java Script Object Notation) is a data format that has been used in a large variety of applications. It is today established as the de-facto standar for representing document stores, for instance, the *MongoDb* database. It is used as well as the request/response format of several RESTful web APIs. Many NoSQL stores have connectors to achieve interoperability through JSON, a role that was previously filled by XML documents.

There are several solutions that aim to provide JSON and NoSQL interoperability. However, most of them try to be generic to support JSON and several other formats as input and also as output, covering data migration issues between NoSQL data sources (Bugiotti et al., 2013). This generality comes with the drawback of implementing integrated frameworks or datamodels not always easy to use.

The approaches can be classified into two main groups. First, the approaches that provide an abstract and generic representation that captures all the constructs of different NoSQL formats, such as (Bugiotti et al., 2013; Atzeni et al., 2014; Alomari et al., 2015). These generic representations act like wrapper structures to access the data sources. The access can be done directly in the original sources or through the translation into the common format. However, it is necessary to maintain the wrapper components or framework throughout the distinct data sources life cycle. In addition, all the sources need to follow the API convention, which may not be always a technical option. Second, many solutions provide translations between specific NoSql Database (Scavuzzo et al., 2014). The translations include a limited number of systems, often between two distinct NoSQL databases. These approaches are more efficient, since they are adapted for specific scenarios. However, their extension requires the implementation of new translations, which may be a costly task. All the given approaches need to store the full object in memory, or to use some lazy loading API. Several other works focus on the migration between RDBMSs and NoSQL, but they are not in the central scope of this paper.

To overcome these issues, we present an approach that focuses on the JSON format as the interoperability data format, and that develops a set of rules to translate to a series of NoSQL formats. We have two main contributions. First, we use the pull-parser programming model (Slomiski, 2001) to read the input JSON objects. The pull-parser programing model has already been used in different scenarios to parse

XML[1] and it has been started to be used with JSON, but not in an interoperability context. This enables to take advantage of well-formed nested JSONs and to read only the parts of the input that are being processed. Second, we provide a set of interoperability rules from JSON to a set of representative NoSQL formats. These rules, which are fully described in the paper, are simple to develop and to extend. They handle 12 NoSQL formats, which cover mostly of the existing representations (Bugiotti et al., 2013).

We validate our approach with an implementation of a prototype that applies the transformations between these data formats, using a public data set as input.

2 RELATED WORK

There are several works aiming to interoperate/convert/migrate/access between different NoSQL databases. We separate them into two major categories.

The first category concentrates on creating wrappers or some kind of homogeneous way to access different data sources, and to translate between the data sources only when necessary. The CDPort framework (Alomari et al., 2015) aims at building a standardized way to access RDBMS and NoSQL Databases though a common data model and an API, both in a cloud-based environment. Each entity can have multiple properties. The different data structures are always accessed with the same primitives. (Michel et al., 2014) proposes a mapping language called xR2RML, to convert heterogeneous data formats to RDF (Resource Description Framework), extending the work from (Consortium et al., 2012) for a NoSql Databases . (Chung et al., 2014) developed a GUI that connects to the column store Hbase. Despite being focused on the translation of queries, the study on the difference of the models also serves to conduct a migration. (Atzeni et al., 2012) presents a programming interface common to NoSql Databases and which can be extended to a RDBMS, called Save Our Systems (SOS). The solution has three main components: a standard interface, one meta-layer responsible for storing the form of the data and specific handlers for each database system. It is the foundation to many other works for uniform data access, including our idea of accessing the databases only through *get()* and *set()* methods. (Scavuzzo et al., 2014) creates a system for migrating data between NoSql columnar databases. He creates a client/server application which uses a metamodel designed solely to handle columnar databases, taking into account details like indexing.

The second major category uses a metamodel, or other kind of intermediate representation, that helps on the NoSQl migration process. The goal is to diminish the number of translation between the data sources, compared to the case of NxN direct translations. (Atzeni et al., 2014) is an extension of the work of (Atzeni et al., 2012), but focusing on the interface utilization. A series of articles present the NoAM (NoSQL Abstract Model) (Bugiotti et al., 2013; Bugiotti et al., 2014; Atzeni et al., 2016), developing solutions based on the observation that the NoSql Databases share similar features, specially the capacity to access their data in what was called "data access units". The classification of representation strategies of this work are the basis for our classification and for the kinds of rules implemented. (Bugiotti et al., 2014) focuses on describing a data modeling and a data design methodology to ensure that the data can be represented in the major NoSql Databases models, and this generic model can be refined or redesigned to better accommodate in the chosen NoSql Databases database. This work is a direct derivate from (Bugiotti et al., 2013) when the database design problem are mainly addressed.

Our approach has two main differences from these previous works. First, it uses JSON as base format, since it is well-established and has many support, without the need to create extra control structures. Second, the input processing and rule execution is done on a stream of objects using the pull-parser programming model, not an API or other similar data access process.

3 JSON-BASED INTEROPERABILITY

In this section we present our approach for JSON-based data interoperability. First, we present how we process the nested JSON format using the pull parser programming model. Second, we describe the migration rules covering different representation strategies.

A JSON document is denoted by the ordered list $JSON = (e_1, e_i, ..., e_n)$, where each element $e_i = (k_i, v_i)$ contains a key k_i and a value v_i, which is either a String s_i, a numeral n_i, a complex object co_i or a collection of elements $C_i = (ec_{i1}, ec_{ij}, ..., ec_{im})$, where each ec_{ij} is itself another element.

Consider the listing below to illustrate the syntax of JSON. The key is the identifier of each element, such as "Person", "firstName" or "type", always in

[1]This model is supported by APIs such as Xerces, kXML, or SAX.

the left side. The elements values, in the right side, may store three kinds of values: 1) simple objects or scalars, such as the String "Smith" or the number 25; 2) complex objects, composed by other objects, such as the "Person" object; 3) collections, such as the "phoneNumber" collection, formed by two elements. This format allows to manipulate and persist a wide diversity of complex values(Hecht and Jablonski, 2011).

```
{ "Person":
  {"firstName":"John","lastName":"Smith","age":25,
  "phoneNumber": [
       { "type": "home", "number": "212 555-1234" },
       { "type": "fax", "number": "646 555-4567" }  ] }
}
```

3.1 Pull-parsing a JSON

The processing of the input JSON elements is done by reading a stream of objects, which means it is not possible to obtain a complete object in advance to store it in memory. We apply the *pull-parser* programming model to read the input objects and to identify its limits and structure. The pull-parser programming model has been used to parse XML documents read from streams in different scenarios. We apply a similar methodology to read JSON input streams.

In this model, the processing algorithm receives a stream of objects $SO = (o_1, o_i, ..., o_n)$, where each object o_i is a tuple $< ek_i, ov_i >$; ek_i is the event kind and ov_i is the object value. The object value is an input JSON element or it can be a NULL value.

The event kinds are separated into four categories: 1) to state the object boundaries (*START_OBJECT*, *END_OBJECT*), 2) to state the boundaries of collections (*START_ARRAY*, *END_ARRAY*), 3) to identify objects (*KEY_NAME*) and 4) to set the object types (*VALUE_STRING*, *VALUE_NUMBER*, *VALUE_TRUE*, *VALUE_FALSE*, *VALUE_NULL*). We adopt the same kind of events supported by the *JSonParser API* [2], since we consider they are enough for many interoperability requirements.

We added the events kinds before each JSON element to illustrate what would be the virtual input of a stream of objects.

```
{START_OBJECT
"Person"KEY_NAME:
  {START_OBJECT "firstName"KEY_NAME: "John"
    VALUE_STRING, "lastName"KEY_NAME:
    "Smith"VALUE_STRING, "age"KEY_NAME: 25
     VALUE_NUMBER,
   "phoneNumber"KEY_NAME : [START_ARRAY
      {START_OBJECT "type"KEY_NAME:
      "home"VALUE_STRING, "number"KEY_NAME:
      "212 555-1234"VALUE_STRING }END_OBJECT,
      {START_OBJECT "type"KEY_NAME:
      "fax"VALUE_STRING, "number"KEY_NAME:
      "646 555-4567"VALUE_STRING }END_OBJECT
   ]END_ARRAY
  }END_OBJECT
}END_OBJECT
```

Every time the application developer calls a **next()** method or function, a new event is processed, which means it is categorized and the input objects are read. The read objects are stored in memory using an intermediate nested data format.

Each object of the intermediate data format stored in memory has the following fields:

ObjectId a unique identifier for each object.

DataValue the value of the given object, if any.

Label the event associated.

FatherObj the ObjectId of the father's object, if any.

The unique identifier is created automatically as a numerical sequence added to each new object. The event is set up as soon as the objects are read. The hierarchy between the objects depends on the existence of collection boundaries events.

The output of the pull parser is illustrated below. It shows the intermediate format after parsing the **phoneNumber** attribute.

```
ObjectId  : 8
DataValue : phoneNumber
Label     : KEY_NAME
FatherObj : 1

ObjectId  : 9
DataValue : null
Label     : START_ARRAY
FatherObj : 8

(the nested objects within
the phone number array)

ObjectId  : 22
DataValue : null
Label     : END_ARRAY;
FatherObj : 9

ObjectId  : 23
DataValue : null
Label     : END_OBJECT;
FatherObj : 1
```

Listing 1: Data format for the phone attribute

It is important to note that these objects are not serialized, but they are processed as soon as they are read from the input stream. The data migration rules follow the sample principle, as it will be shown in the next section.

[2] http://docs.oracle.com/javaee/7/api/javax/json/stream/JsonParser.html

3.2 Interoperability Rules

The interoperability rules developed take into account the representation strategies presented in (Bugiotti and Cabibbo, 2013), since they cover a large number of NoSQL representations. We separate the rule description by the category of input data model and we illustrate the output of each rule execution. The execution of each rule is illustrated by using the "Person" element already presented[3].

Each rule is fired once a new object is identified, i.e., a START_OBJECT event occurs. For each execution, the rules process the following properties:

- *Class*: The class name defines the identifier of a given composed object [4]. This means that all the nested objects or arrays have the same kind. In the Document Store model, the class name is called *Collections*; in the Graph model the class name is the *main node*.
- Key: each object will have a main key, according to the data model properties.
- *Value*: the value indexed by a given MainKey.

The difficulty on specifying the rules may vary depending on the output data model. For instance, in some cases it is more difficult to produce the output key than the output data, or vice-versa. This will be clearer in the following sections.

3.3 Key-Value stores

A key-value store contains collections of key-value (K,V) pairs, where the key K is used as an index to perform operations over the value V.

Key-value per object - kvpo: there is only one object associated per each key. The key is a concatenation of the collection name and an identifier for the object. The collection name could be considered the object type. The value is a serialization of the entire value of the object, which may be a atomic data type or a composition of values or objects.

The MainKey that identifies an object is formed by the object *Class* plus the first VALUE_STRING found. The Value is generated by concatenating all the nested values of the object. The output is a sequence of key-values pairs, as shown in Table 1.

Key-value per field - kvpf: there are multiple key-value pairs to represent each object. The key is a concatenation of the collection name, the object identifier and the name of the top-level field. The format

Table 1: Key-value per object - kvpo()

Key	Value

MainKey	**for all** Obj.value **do**
	Value = *Value* + Obj.value
	end for

Person:John	"firstName":"John", "lastName": "Smith", "age": 25, "phoneNumber": [{ "type": "home", "number": "212 555-1234" }, ...]

of the key may vary depending of the implementation, keeping the requirement that the value is only the value of the corresponding field.

The MainKey is the object *Class* plus the KEY_NAME, and this is repeated for each KEY_NAME found in the input object. The value is the data associated at the KEY_NAME. If the data is an Array or other Object all the values are concatenated until the end of the Array or Object (see Table 2).

Table 2: Key-value per field - kvpf()

Key	Value

MainKey + "/" + Obj.KEY_NAME	**for all** Obj.KEY_NAME **do**
	if *Value* = (*Array* **or** *Object*) **then**
	for all Obj.value **do**
	Value = *Value* + Obj.value
	end for
	else
	Value = Obj.value
	end if
	end for

Person:John/	John
firstName	Smith
lastName	25
age	{ "type": "home", "number": "212
phoneNumber	555-1234" }, ...

Key-value per field object - kvpfo: the key is a concatenation of a major and a minor key. The major key contains information related to the main object, such as its collection name and an identifier and the minor key has information related to each field.

The Key is composed by the MainKey , plus /-/, plus each KEY_NAME found in the object. The values are formed by the KEY_VALUE associated to the KEY_NAME. If the the value is an array or other object, it is sequentially concatenated (3).

Key-value per atomic value - kvpav: the key is a concatenation of identifiers, and the value is a unique

[3]We removed the second phone number in the illustrations for brevity

[4]In this work a class is used as a noun to categorize an object with a set of common attributes

Table 3: Key-value per field object - kvpfo()

Key	Value

MainKey + ”/-/” + Objs.KEY_NAME	**for all** Obj.KEY_NAME **do**
	if *Value* = (*Array* **or** *Object*) **then**
	for all Obj.value **do**
	Value = *Value* + Obj.value
	end for
	Value = Objs.KEY_NAME + ”:” + *Value*
	end if
	Value = Objs.KEY_NAME + ”:” + Obj.Value
	end for

Person/John/-/firstName Person/John/-/lastName Person/John/-/age Person/John/-/phoneNumber	John Smith 25 ”type”: ”home”, ”number”: ”212 555-1234” , ...

Table 4: Key-value per atomic value - kvpav()

Key	Value

for all Obj.KEY_VALUE **do**
Key = MainKey + ”/-/” + Objs.KEY_NAME
if *Value* = (*Array* **or** *Object*) **then**
for all Obj_i **do**
Key = *Key* + ”/” + Obj_i + Obj.KEY_NAME
end for
end if
end for
	Objs.KEY_NAME. Value

Person/John/-/firstName Person/John/-/lastName Person/John/-/age Person/John/-/phoneNumber/0/type Person/John/-/phoneNumber/0/number Person/John/-/phoneNumber/1/type Person/John/-/phoneNumber/1/number	John Smith 25 home 212 555-1234 fax 646 555-4567

Table 5: Key-hash per object - khpo()

Key	Value

MainKey	**for all** Obj.KEY_NAME **do**
	if *Value* = (*Array* **or** *Object*) **then**
	for all Obj.value **do**
	Value = *Value* + Obj.value
	end for
	Value = Objs.KEY_NAME + ”:” + *Value*
	end if
	Value = Objs.KEY_NAME + ”:” + Obj.Value
	end for

Person:John	firstName:John lastName:Smith age:25 phoneNumber:[”type”: ”home”, ”number”: ”212 555-1234” , ...]

atomic value, not allowing complex objects.

The values are formed by each of the KEY_VALUE's found. The Key is composed by the MainKey , plus /-/, plus all the path until the KEY_NAME before the value. If the value is an array or another object, a sequential number is added in the key to maintain the uniqueness (see Table 4).

Key-hash per object - khpo: there is a key for each complex object and a *hash* for each field value, which is commonly the field value.

The Key has the same format of the kvpo representation. The same MainKey has several vales, each one composed by the KEY_NAME plus the associated value. If the value is an array or other object, the value is the concatenation of all elements of the array or object (see Table 5).

3.4 Column Stores

Column Stores are organized on columns (as its central entity), tables and rows. Thus, they are optimized for reading columns, or groups of columns.

Column: a *Column* organizes keyed records as a collection of columns, where a column contains collections of key-value pairs. The key is the column name, and the value can be an arbitrary data type.

The column name is each individual KEY_NAME and the values are formed by each of the individual KEY_VALUE's. If the value is an array or other object, the columns' name are composed by the KEY_NAME of the father plus the final KEY_NAME found. No group is created, and the columns are stored individually (see Table 6 (a)).

Super Column: it is a collection containing records of other columns, so each column is a group of other columns, and these groups are stored and manipulated based on a ”Super Column” name, which can be defined as a Key part, and the columns group

itself determine the value.

The migration rule is a variation of the previous one. The identification of the key is the same, as well as the assignment of the values. The rule changes when the value is an array or another object: the KEY_NAME of the father object is used as a Super Column name, with the other KEY_NAME's serving as the column name (see Table 6 (b)).

Table 6: Column and super column rules

(a) Column

Column	Value

for all Obj.KEY_NAME **do**
if Obj.hasFather = **true** **then**
Key = Obj_f.KEY_NAME + "/" + Obj.KEY_NAME
else
Key = Obj.KEY_NAME
end if
end for
	Obj.KEY_VALUE

firstName	John
lastName	Smith
age	25
phoneNumber/type	home
phoneNumber/number	212 555-1234
phoneNumber/type	fax
phoneNumber/number	646 555-4567

(b) Super Column

Super Column	Column	Value
Obj_f.KEY_NAME	KEY_NAME	KEY_VALUE
phoneNumber	firstName	John
	lastName	Smith
	age	25
	type	home
	number	212 555-1234
	type	fax
	number	646 555-4567

Column Family: it groups the columns based in a Row Key, which is set by the first VALUE_STRING found (see Table 7 (a)). The creation of the columns follow the creation rules of a Super Column.

Super Column Family: the Row Key groups columns that are correlated. The Row Key is set by the object *Class*, which plays a role similar of a table name. The columns follow the creation rules of a Super Column. The rule is shown in Table 7 (b).

3.5 Document Stores (DS)

The document stores are designed to manipulate and persist a wide diversity of complex values (Hecht and Jablonski, 2011), which can comprise scalar values, lists, and other documents in a nested format. These documents are organized into collections of objects, i.e., a group of documents.

Table 7: Column Family and super column family

(a) Column Family, row key 'John'

Super Column	Column	Value
Obj_f.KEY_NAME	KEY_NAME	KEY_VALUE
phoneNumber	firstName	John
	lastName	Smith
	age	25
	type	home
	number	212 555-1234
	type	fax
	number	646 555-4567

(b) Super Column Family, column family 'Person'

Super Column	Column	Value
Obj_f.KEY_NAME	KEY_NAME	KEY_VALUE
phoneNumber	firstName	John
	lastName	Smith
	age	25
	type	home
	number	212 555-1234
	type	fax
	number	646 555-4567

Similarly to Key-Value stores, there are variations on how to encode the documents. The three main variations are **document per object - cpo**, **item per object - ipo** and **cell per object - cpo**.

The migration rules have similarities to the Key Value stores, since the objects may be identified by unique keys. We describe the particularities in the following.

Document per object: the migration rule is similar to the kvpo strategy. The main difference is that the MainKey is split into the class name, acting as a collection name and the first VALUE_STRING, acting as the "Document id". The nested values are concatenated sequentially. This rule is described in Table 8.

Table 8: Document per object - dpo(), class Person

Document id	Value

VALUE_STRING	**for all** Obj.value **do**
	Value = *Value* + Obj.value
	end for

John	{"firstName":"John", "lastName": "Smith", "age": 25, "phoneNumber": { "type": "home", "number": "212 555-1234" }, ...

Item per object: this rule is similar to the kvpf one. The class name is the Collection name and the data is composed by the KEY_NAME and the associ-

ated value. To distinguish each collection within the same element, one ID is generated for each inner document. If the value is an array or other object, it is the concatenation of all the nested elements (see Table 9).

Table 9: Item per object - ipo(), class Person

Documents	Value
KEY_NAME	**for all** Obj.KEY_NAME **do** *Value* = *Value* + Obj.value **end for**
_id	
firstName	John
lastName	John
age	Smith
	25
phoneNumber	{ "type": "home", "number": "212 555-1234" }, ...

Cell per object: the table name receives the *Class name*. The ID is created based on the first VALUE_STRING found. The Value receives all the nested values concatenated sequentially (see Table(10).

Table 10: Cell per object - cpo(), class Person

Value	
VALUE_STRING	**for all** Obj.value **do** *Value* = *Value* + Obj.value **end for**
John	John {"firstName":"John", "lastName": "Smith", "age": 25, "phoneNumber": [{ "type": "home", "number": "212 555-1234" },]}

3.6 Graph stores

A graph store organizes the data as nodes, edges and properties. Is important to note that the properties are key/values pairs. Nodes can represent entities, and the edges are the connection between two nodes representing a relationship and the properties are the data itself (Bondiombouy and Valduriez, 2016). There are several possible representations, such as not considering properties as separate entities as well. They are best suited to applications involving large connected elements, graph traversals and sub-graph matching.

The Main Node is composed by the object *Class*, plus the first VALUE_STRING found. This is the same process used to form the MainKey . The leaf nodes are composed by each KEY_NAME, plus the associated value. If the value is an array or another object, it is the concatenation of all elements of the array or object (see Table 11). Note that graph databases may have many other encoding, which are not covered by this migration rule.

Table 11: Graph - graph(), node Person

Leaf Node	Value
Objs.KEY_NAME	**for all** Obj.KEY_NAME **do** **if** *Value* = (*Array* **or** *Object*) **then** **for all** Obj.value **do** *Value* = *Value* + Obj.value **end for** *Value* = Obj.KEY_NAME + ":" + *Value* **end if** *Value* = Obj.KEY_NAME + ":" + Obj.Value **end for**
firstName	John
lastName	Smith
age	25
phoneNumber	{ "type": "home", "number": "212 555-1234" }, ...

3.7 Implementation

The implemented tool[5] uses different NoSQL databases per category of data store. They where chosen because they have all implemented **get()** and **put()** interfaces to access the data, as well as ways to serialize the results in JSON. As Key value store, we use the Oracle NoSQL Community Edition; for the column stores, Apache HBase; Mongo Db as document store and Neo4J as graph database.

We used the data that is freely available from the City of Chicago Data Portal and the "Food Inspections" data set [6]. The dataset describes inspections of restaurants and other food establishments in Chicago from January 1, 2010 to December 1, 2016. There is no particular reason about the kind of data chosen, just because they are public domain, with easy access through its API. The input data contains 139.535 objects. Each object is composed by 23 fields and 1 array of objects, containing itself 5 distinct fields. Table 12 shows the number of output pairs for each representation strategy for key value stores.

[5]http://www.inf.ufpr.br/didonet/files/Jsonpullparser.zip

[6]Food Inspections Data Set: https://data.cityofchicago.org/Health-Human-

Table 12: Generated elements for Key Value stores

	MainKey	Values	Output Pairs
Kvpo	1	1	139.535
Kvpf	24	24	3.348.840
Khpf	1	24	3.348.840
Kvpfo	24	24	3.348.840
Kvpav	28	28	3.906.980

For Column Stores, it generates the same number of columns as output, 3.906.980, for Column, Super Column, Column Family and Super Column Family. The output is different only in the way the columns are grouped. For the Document Stores, the choice of the key that will compose the document has a direct consequence in the number of generated values: *dpo* produced 139.535 elements; *ipo* generated 3.348.840 and `cpo` generated 139.535 elements. Finally, the output for the Graph databases was one main node, the input class, and one leaf node for each field or array in the original file. The values are then inserted into each leaf node, totalling 3.348.840 elements.

4 CONCLUSIONS

We presented an approach for NoSQL interoperability based on the JSON format and applying the pull-parser programming model for executing a set of rules over a stream of objects. We have two main contributions. First, we use the JSON nested data model as a basis for interoperability between different NoSQL data formats. The utilization of JSON has confrimed to be an effective choice, since it has many support for several APIs, making it easy to connect to different output datastores.

The second main contribution is the utilization of the pull-parser programming model, which has already been used in the XML context, for reading the input from a stream of objects. This enables to have large files as input, since it does not need to keep the input objects in memory. The translation itself is free of context, if the JSON objects are well-formed nested documents.

We detailed a set of rules from JSON to a set of NoSQL data representation strategies. The data migration rules are simple to implement, relying only on *get()* and *set()* primitives, available in several implementations of NoSQL databases. Despite covering a large number of representations,other representations exist, specially with respect to the composition of the input keys. They are often path/based expressions to reach a given object.

Services/Food-Inspections/4ijn-s7e5

As future work, we could extend the model to support complex query compositions, and to compare the results of a same query in different NoSQL stores.

REFERENCES

Alomari, E., Barnawi, A., and Sakr, S. (2015). Cdport: A portability framework for nosql datastores. *Arabian Journal for Science and Engineering*, pages 1–23.

Atzeni, P., Bugiotti, F., Cabibbo, L., and Torlone, R. (2016). Data modeling in the nosql world. *Computer Standards & Interfaces*.

Atzeni, P., Bugiotti, F., and Rossi, L. (2012). Uniform access to non-relational database systems: The sos platform. In *Advanced Information Systems Engineering*, pages 160–174. Springer.

Atzeni, P., Bugiotti, F., and Rossi, L. (2014). Uniform access to nosql systems. *Information Systems*, 43:117–133.

Bondiombouy, C. and Valduriez, P. (2016). *Query Processing in Multistore Systems: an overview*. PhD thesis, INRIA Sophia Antipolis-Méditerranée.

Bugiotti, F. and Cabibbo, L. (2013). A comparison of data models and apis of nosql datastores. *Dipartamento di Ingegneria della Università di Roma*.

Bugiotti, F., Cabibbo, L., Atzeni, P., and Torlone, R. (2013). A logical approach to nosql databases.

Bugiotti, F., Cabibbo, L., Atzeni, P., and Torlone, R. (2014). Database design for nosql systems. In *In proc. of ER*, pages 223–231. Springer.

Chung, W.-C., Lin, H.-P., Chen, S.-C., Jiang, M.-F., and Chung, Y.-C. (2014). Jackhare: a framework for sql to nosql translation using mapreduce. *Automated Software Engineering*, 21(4):489–508.

Consortium, W. W. W. et al. (2012). R2rml: Rdb to rdf mapping language.

Hecht, R. and Jablonski, S. (2011). Nosql evaluation: A use case oriented survey.

Michel, F., Djimenou, L., Faron-Zucker, C., and Montagnat, J. (2014). xr2rml: Relational and non-relational databases to rdf mapping language. Technical report, ISRN I3S/RR 2014-04-FR v3.

Scavuzzo, M., Di Nitto, E., and Ceri, S. (2014). Interoperable data migration between nosql columnar databases. In *2014 IEEE 18th EDOCW*, pages 154–162. IEEE.

Slomiski, A. (2001). TR550: Design of a Pull and Push Parser System for Streaming XML. Technical report, University of Indiana, US.