International Journal of Engineering & Technology, 7 (4.32) (2018) 11-16

International Journal of Engineering & Technology

Website: www.sciencepubco.com/index.php/IJET

Research paper



A Logical Approach for Real Time Big Data Analytics on Heterogeneous Nosql Databases

Omar HAJOUI[1]*, Mohamed Talea[2]

[1,2]LTI Laboratory, Faculty of Science Ben M'sik Hassan II University, Casablanca, Morocco
**Corresponding author E-mail: hajouio@yahoo.fr*

Abstract

NoSQL databases are developed to provide a set of new big data management features while overcoming certain limitations of relational databases. However, these databases are heterogeneous; they provide different mechanisms for storing and retrieving data, which directly affect the performance, consistency and availability of data. In addition, they offer different models of data storage, different implementations, query languages and APIs. This wide variety of platforms makes data interoperability difficult. Data interoperability can be defined as the ability of an application to interact at the same time with a set of different and heterogeneous systems. The goal of our research is to design a new approach that makes it easy for applications to analyze and explore data stored in multiple NoSQL databases. Our approach is based on a Meta model for transforming data from one model to another. Also, we have developed a common API that hides the access specificities of each NoSQL database while allowing the transformation of this data into JSON format.

Keywords: *Big data analytics; Interoperability; NoSQL databases; Meta model; Extraction; Transformation.*

1. Introduction

Since there are different types of NoSQL databases (key-value, document store, columnar store, graph database) that have different performance and different behavior in terms of availability and consistency, some companies are not satisfied with the use of a single NoSQL database. Generally large applications, such as Facebook, Google, LinkedIn, Amazon etc., require the use of multiple NoSQL databases. Sometimes every part of this kind of application uses a NoSQL database. This popular case known as polyglot persistence [1], refers to the fact that an application uses several NoSQL databases simultaneously to meet the needs of the company.

Unfortunately, writing the code of an application that uses multiple databases is not easy and sometimes very constraining for the developer. For example, in the case of relational databases, just use the JDBC API to access a MySQL database, Oracle and others. For cons, the lack of a standard for accessing NoSQL databases is a great problem, as each NoSQL database has its own data model, query language and APIs. This kind of problems related to the heterogeneity of NoSQL databases has been evoked for the first time by Stonebraker [2].

The real value of big data is in the insights it produces when analyzed—discovered patterns, derived meaning, indicators for decisions, and ultimately the ability to respond to the world with greater intelligence. Big data analytics is a set of advanced technologies designed to work with large volumes of data. It uses sophisticated quantitative methods such as machine learning, neural networks, robotics, computational mathematics, and artificial intelligence to explore the data and to discover interrelationships and patterns [3]. But when data of a company or an organization is managed by heterogeneous database management systems (Redis, Mongodb, Cassandra, Hbase, Voltdb, Couchbase, Memcached, etc.), querying and data analysis become very difficult. So we face the problem of data interoperability [4].

Big data interoperability problem is produced when we want to exploit and analyze data stored in heterogeneous systems. Storing data in different representations makes data querying and interpretation difficult. For example, this problem arises if we want to make statistics on the data stored in several institutions of the same organization, several subsidiaries of the same company, several commercial sites of the same mall or intelligent sensors in the same city.

The goal of this work is the design and implementation of a new approach that facilitates the interoperability of NoSQL databases, in particular key-value store, document store and columnar store. This approach is based on the following principle: to develop an application that queries or analyzes data distributed across multiple NoSQL databases (data sources). We start by integrating this data into another target NoSQL database to prepare it for querying. Since data sources can have different data models (key-value store, document store or columnar store). This phase requires data transformation to the model of the target database before data integration in the same data warehouse. To make the transformation simple and efficient we have adopted the HashMap data model as a transformation metamodel. Thus, initially, the source data is transformed to our metamodel and then to the data model of the target database. Given N different models, the direct transformation from one model to the other requires N * (N-1) transformations in place of 2 * N with the use of a metamodel. Once data integration is done, applications and users can query the target database directly. To validate our approach, we have developed a framework that allows transformations from source databases with different data models to the HashMap metamodel: Redis (key-value store), Cassandra (columnar store), and MongoDB (document store).

Copyright © 2018 Authors. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

This paper is organized as follows: section 2 presents different approaches for big data interoperability. In Sections 3, 4, 5, 6 and 7 , we present a new approach for real time big data interoperability. Section 8 provides a conclusion and some open issues.

2. Related Work

Currently, there are several works that address the big data interoperability. These approaches can be classified into 4 categories: framework [5, 6,7], generic API [8, 9, 10,11], federated approach [12, 13] and model-driven architecture based approach [14].

According to the results of our previous study [15], ONDM is the best approach for big data interoperability. This approach perfectly solves the problem of heterogeneous data models by NoAM (NoSQL Abstract Model) [5], having a good performance, flexible and can be easily used in a cloud environment. It remains to improve this framework to run complex queries such as aggregates functions.

The federated approach has the advantage of introducing a SQL-like query interface, making it easy the execution of complex queries. But this solution is strongly related to NoSQL databases used for data storage. The modification or addition of a new database causes the rewriting of the solution code.

ODBAPI is a good approach, implemented in a cloud environment and flexible. However, it does not provide a complete solution to the data models heterogeneity problem and its performance has not been evaluated.

Although these works facilitate access to several NoSQL databases through a single application. Some are more effective than others. Except federated approaches that adopt a language similar to SQL, most approaches allow only the CRUD operation. the generalization of CRUD operations facilitate data integration but do not provide an effective solution to the interoperability problem that require the execution of more complex queries (e.g. GROUP BY, LIKE, JOIN, etc.). This kind of complex queries is at the base of Big Data analytics. Some did not take into account other categories of data stores such as column data stores and graph data stores, some are implemented in a cloud environment other none.

3. An Approach for Big Data Analytics on Heterogeneous Nosql Dbaas

Our approach is based on the model of the figure below

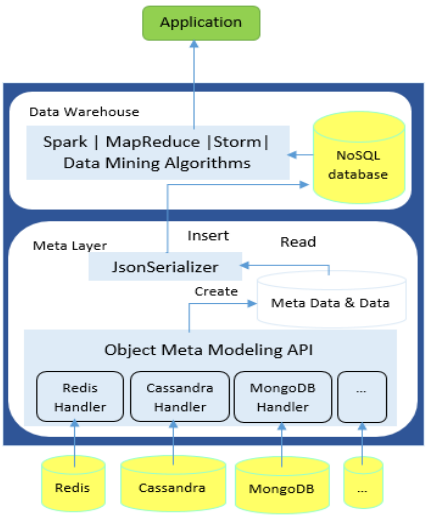

Fig. 1: approach model

This model can be divided into two main layers:

Meta Layer: the data we want to analyze comes from several databases, each one has its own data storage model (key-value store, document store, columnar store). The role of this layer is to solve the problems related to the heterogeneity of NoSQL databases including the heterogeneity of data models. To do so, this layer adopts a logic based on a Meta model. A Meta model is a generic model to which data from different data sources will be translated. After this phase of transforming the data from the source databases to a Meta model, comes the integration phase of the data into a target database which is our Data Warehouse.

Data Warehouse (DW): A database used to collect, integrate data from operational databases, and provide a decision support. This data can be analyzed either by the query language specific to this database, visualization or reporting tools, predictive algorithms, or by applications using the MapReduce Framework, Apache Spark or other APIs.

4. Meta Layer

As we have already mentioned, the role of this layer is to solve the problem of heterogeneity of NoSQL databases. This requires the extraction of data from the source databases and the transformation of these data to the data model of the target database (data warehouse) or to a Meta model. In our approach, we have opted for the transformation to a Meta model: suppose we have N source databases, the transformation to a Meta model or to the model of a target database requires N transformations. In the second case, a possible change of the target database causes N other transformations to the new database. On the other hand, in the first case, the addition of a single transformation from the Meta model to the new target database is necessary. This makes the extraction and transformation APIs of the Meta Layer independent of the representation of data in the data warehouse.

4.1. The Meta model

We adopted the data model of a hash table as a Meta model. A hash table (hash map) is a data structure which implements an associative array abstract data type, a structure that can map keys to values. Each element of the table is accessed by its key. So our meta model (Figure 2) can be seen as a set of entries, each entry has a key and a value, the key is usually a string, the value can be of elementary type (int, float, string, date, boolean), a list or HashMap. We chose the data model of a hash map like Meta model because most programming languages implements this structure. In addition, they provide APIs that make it easy to load data into a hash table.

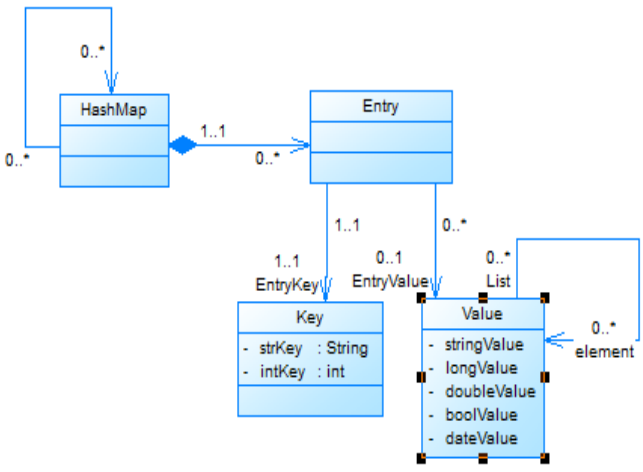

Fig. 2: The Meta model

4.2. Transformation rules

Once the Meta model is defined, we need to establish a set of transformation rules to perform the transformation from different NoSQL data models to our Meta model. NoSQL databases can be classified according to four categories: key-value store, columnar store, document store and Graph database. In this work, we only considered the first three categories. The table below shows a comparison between the different data models:

Table 1: A comparison between different data models

Relational concepts	Key-value store	Columnar store	Document store	Meta model
Table	Kind name	Column Family	Collection	HashMap
Primary Key	Kind : id	RowId	DocumentId	key
Row	value	Row	Document	value
Column	field	Column	field	field

In the following, we present the transformation rules from the different models of NoSQL databases to our Meta model, as well as our transformation algorithms.

4.2.1. Transformation rules from Key-Value store

In key-value store (Figure 3), data is simply represented by a key / value pair, both the key and the value can be of any structure, their model can be likened to a distributed hash table. The key must be self-descriptive with respect to a schema. For example "object-type: id" is a good idea, as in "Patient: 01". The value can be of type: Binary-safe strings, Lists, Sets, Sorted sets, Hashes.

key	value
"Patient : 01 : FirstName"	"Mohammad Ali"
"Patient : 01 : LastName"	"Clay"
"Patient : 01 : Gender"	"M"
"Patient : 01 : DateOfBirth"	"1942-01-17"
"Patient : 01 : PlaceOfBirth"	"Louisville, Kentucky"
"Patient : 01 : PlaceOfResidence"	"Phœnix, Arizona"
"Patient : 01 : idC "	200
"Patient : 01 : idC "	363
"Consultation : 200 :disease "	"Parkinson"
"Consultation : 200 : drFirstName "	"Ferdie"
"Consultation :200 : drLastName "	"Pacheco"
"Consultation :200 :dateOfConsultation "	"1996-09-04"
"Consultation :200 :treatment "	"L-Dopa"
"Consultation :200 :treatment_efficiency "	true
"Consultation :363 :disease "	"migraine"
.....	.....

Fig. 3: Key-Value store example

The transformation from a Key-Value database to a hash table is quite simple (Algorithm 1), as both structures have the same data model. For example, if you want to prepare patient data for analysis. We need to extract all patient-type entities to load into a new Hash Map named dataHashMap. In this case, the keys of the dataHashMap will be the identifiers of the patients (we extract the value after the first ':'), the values are of type Hash Map. The value that corresponds to each patient is a Hash Map whose keys are the properties of the patient (the fields after the second ':') and the values are the values of these fields. Thus, one can have several nested Hash Map.

Algorithm 1: Translation from Key-value data model to HashMap

Condition: dataHashMap, metadataHashMap must be instancied, the database exists

1. **method extract(databaseName ,entityName)**
2. entityKeys <- getKeysWithEntityName(databaseName , entityName)
3. foreach key in entityKeys
4. keyId <- extractId(key)
5. attributesHashMap <- extractAttributesAndValues (keyId,databaseName)
6. dataHashMap.Add(keyId,attributesHashMap)
7. endforeach
8. metadataHashMap <- extractMetaData(databaseName , entityName)
9. **end method**

Patients : HashMap

Key	value		
01	key	value	
	FirstName	"Mohammad Ali"	
	LastName	"Clay"	
	Gender	"M"	
	DateOfBirth	"1942-01-17"	
	PlaceOfBirth	"Louisville, Kentucky"	
	PlaceOfResidence	"Phœnix, Arizona"	
	Consultations		
	Key	value	
	200	key	value
		disease	"Parkinson"
		drFirstName	"Ferdie"
		drLastName	"Pacheco"
		dateOfConsultation	"1996-09-04"
		treatment	"L-Dopa"
		treatment_efficiency	true
	363	.....	
02	.....		

Fig. 4: the result of algorithm 1 after the transformation

4.2.2. Transformation rules from columnar store

The columnar store type (Figure 5), are conceptually quite close to the relational tables (Column Family): they contain columns with a data type and Super Columns. Each line of this table has a unique key. Inside the table, Super Columns are defined and group together a set of columns. The Super Column is predefined, while the columns in it are not predefined. From one line to another, the columns in a Super Column may vary depending on the data stored.

Patients

RowId	Infos	Consultations []
2001	FirstName="Ray" LastName="Robinson" Gender="M" DateOfBirth="1921-25-03" PlaceOfBirth="Detroit, Michigan" PlaceOfResidence="Manhattan,New York"	[0].id=100 [0].disease=" Alzheimer" [0].drFirstName="Dennis " [0].drLastName ="Cope" [0].dateOfConsultation="1964-05-12" [0].treatment="Alzen" [0].treatment_efficiency=false
		[1].id=153 [1].disease="migraine" ....
2002	.....	.....

Fig. 5: The column-oriented database example

So, if we do the mapping between the columnar store model and our Meta model (HashMap). The big table (Column Family) is a HashMap. The keys of the HashMap are only the identifiers of the rows of the table. At this level, the value that corresponds to each key is another HashMap whose keys are the names of the columns of the same row and the values are the values of these columns. If the column is a Super Column (groups several columns). In this case, the key is the name of the group and the value is a HashMap with the names of columns as keys and their values as values. Algorithm 2 shows our transformation logic from the columnar store model to our Meta model.

Algorithm 2: Translation from Columnar data model to HashMap

Condition: dataHashMap , metadataHashMap must be instancied, the database exists

1. **method extract(keyspaceName,tableName)**
2. rows <- getRows(keyspaceName,tableName)
3. foreach row in rows
4. columnsHashMap <- extractColumnsOrSuperColumnsValues (row)
5. dataHashMap.Add(rowId,columnsHashMap)
6. endforeach
7. metadataHashMap <- extractMetaData(keyspaceName,tableName)
8. **end method**

4.2.3. Transformation rules from document store

A document store database (Figure 6) consists of a set of collections. Each collection is a set of JSON documents. Each document is identified by an ID. Documents do not necessarily have the same structure.

```
[
{
"idP": 100,
"FirstName": "Mary",
"LastName": "Jane",
"Gender": "F",
"DateOfBirth": new Date ('Jun 23, 1970'),
"PlaceOfBirth": "Omaha, Nebraska",
"PlaceOfResidence": "Omaha, Nebraska",
"Consultations": [
        {"id": 20, "disease": "diabetes", "drFirstName": "John", "drLastName": "Bradley",
        "DateOfConsultation": new Date ('Feb 23, 1992'), "treatment": "insulin",
        "treatment_efficiency": true},
        .....
            ]
}
{
"idP": 101,
.....
}
]
```

Fig. 6: document database example

We will also describe the mapping rules between the document store model and our Meta model (HashMap). Each collection corresponds to a HashMap. The identifier of each document in the collection is a new key in the HashMap. The value that corresponds to each key is another HashMap whose keys are the names of the fields of the current document and the values are the values of these fields. If the field is composed of several other fields, its value is a HashMap whose keys are the names of the component fields and the values are the values of these fields. If the value of the field is an array, we build a list whose each line is a HashMap that contains the fields and their values. Thus, we make a recursive call to scan all nesting, but in general, in our experience, the number of nestings does not exceed 3 from the root of each document. Algorithm 3 shows our transformation logic from the document store model to our Meta model.

Algorithm 3: Translation from Document data model to HashMap

Condition: dataHashMap , metadataHashMap must be instancied, the database exists

1. **method extract(databaseName,collectionName)**
2. DBCursor <- collectionName.find()
3. While(DBCursor.hasNext())
4. fieldsHashMap <- extractFieldsAndValues(DBCursor.next())
5. dataHashMap.Add(documentId , fieldsHashMap)
6. endwhile
7. metadataHashMap <- extractMetaData(databaseName, collectionName)
8. **end method**

4.2.4. Serializing the contents of a HashMap in a Document Database

After data extraction and transformation comes the step of integrating data into a target database. So we have to load the data from our Meta model (HashMap) to the model of the target database. In our implementation, we chose the MongoDB database which is a document store based like a data warehouse. We justify this choice in detail in the following section. In addition, MongoDB provides an API that allows the automatic serialization of a JAVA object in JSON format. This makes it quite simple to integrate the contents of a HashMap into a MongoDB collection (Algorithm 4).

Algorithm 4: Translation from HashMap to Document store

Condition: the database exists

1. **method serialize(databaseName,collectionName)**
2. DB db = mongo.getDB(databaseName)
3. DBCollection collection=db.getCollection(collectionName)
4. foreach value : HashMap in dataHashMap.values
5. AddDocument(collection value)
6. end foreach
7. **end method**

5. Data Warehouse

The data warehouse is a NoSQL database that has two main roles:

- Integration of data collected from different sources
- Data analysis

For data integration, any NoSQL database can fulfill this role, our approach is independent of the target database. But the need to analyze this data pushes us to choose this database carefully. Especially in Big Data, the analytic part becomes more and more important. For advanced and real-time big data analytics, the best framework you can use is Apache Spark [16]. According to the official version, Spark uses the Hadoop HDFS file system.

Spark offers a complete and unified framework to meet the needs of real time big data analytics for various datasets, various by their nature (text, graph, etc.) as well as by the type of source (batch or real-time flow). It allows to quickly write applications in Java, Scala or Python and includes a set of more than 80 high-level operators, it is possible to use it interactively to query the data from a shell, in addition to the operations of Map and Reduce, Spark supports SQL queries and data streaming and offers machine learning and graph-oriented processing functions. Developers can use these possibilities in stand-alone or by combining them into a complex processing chain.

Once the data is stored in the Hadoop file system (HDFS), the Hadoop ecosystem offers other sophisticated analysis tools: MapReduce Framework, Pig & Hive that offer a language similar to SQL, Mahout for machine learning algorithms, cloudera Impala. Thus, we chose Hadoop ecosystem as data warehouse in our approach.

6. A JAVA API for NoSQL Databases Interoperability

To test our approach, we developed a JAVA API that facilitates the interoperability of NoSQL databases (Figure 7). This API was developed in a cloud platform specific to each system (redislabs for Redis, datastax for Cassandra, mLab for MongoDB)

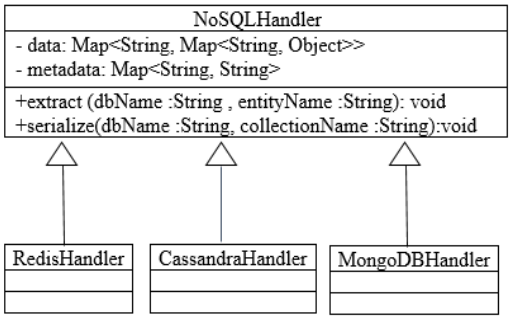

Fig 7: An API for NoSQL databases interoperability

In this API, the super class is the NoSQLHandler class , this class contains two methods:

- **The extract method**: at this level this method is abstract, its code is redefined in the RedisHandler , CassandraHandler and MongoDBHandler classes. This method receives as parameters the database path and the name of the entity (global key for Redis, a family of columns for Cassandra, a collection for MongoDB). The code of this method extracts all data from the database entity given in parameter and loads them into the HashMap data. The HashMap metadata contains information about the entity name and the names of the columns and their types. In section V.B of this chapter, we presented and explained in detail the transformation algorithms used in this method for each type of database.

- **The serialize method**: this method makes it possible to serialize the contents of HashMap data in a MongoDB collection whose name is given in parameter. The serialization algorithm used in this method is presented in section V.B the MongoDB BasicDBObject class makes it easier to serialize a HashMap into a collection.

7. Case study

To test the validity of our approach, we applied it to a case study inspired by the field of health. In this case study, the management of a group of hospitals wants to analyze data stored in several sites. The first hospital implements the Redis system (Key-Value store), the second the Cassandra database (Columnar store) and the third uses the MongoDB database (Document store). Figure 8 presents the data model of this group.

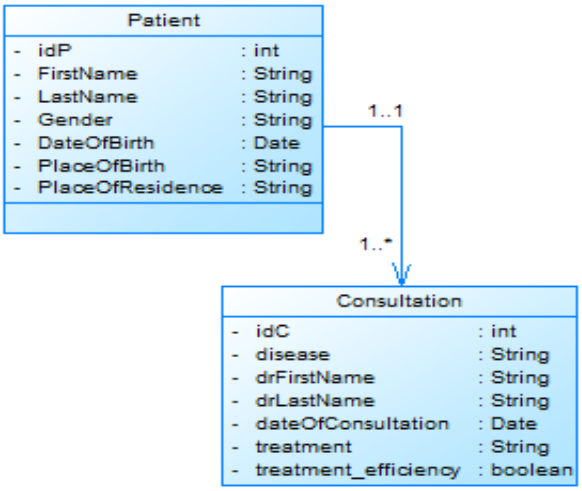

Fig. 8: case study (group of hospitals)

Management can issue requests such as:

- The number of people affected by a particular disease?
- The list of effective treatments for a given disease?
- The patient consultation history at the group level?

It is obvious that these requests require the interrogation of the three systems of the group

Figures 9, 10, 11 show an extract from the dataset of each database:

key	value
"Patient : 01 : FirstName"	"Mohammad Ali"
"Patient : 01 : LastName"	"Clay"
"Patient : 01 : Gender"	"M"
"Patient : 01 : DateOfBirth"	"1942-01-17"
"Patient : 01 : PlaceOfBirth"	"Louisville, Kentucky"
"Patient : 01 : PlaceOfResidence"	"Phœnix, Arizona"
"Patient : 01 : idC "	200
"Patient : 01 : idC "	363
"Consultation : 200 :disease "	"Parkinson"
"Consultation : 200 : drFirstName "	"Ferdie"
"Consultation :200 : drLastName "	"Pacheco"
"Consultation :200 :dateOfConsultation "	"1996-09-04"
"Consultation :200 :treatment "	"L-Dopa"
"Consultation :200 :treatment_efficiency "	true
"Consultation :363 :disease "	"migraine"
.....	.....

Fig. 9: RedisLabs database

Patients		
RowId	Infos	Consultations []
2001	FirstName="Ray" LastName="Robinson" Gender="M" DateOfBirth="1921-25-03" PlaceOfBirth="Detroit, Michigan" PlaceOfResidence="Manhattan,New York"	[0].id=100 [0].disease=" Alzheimer" [0].drFirstName="Dennis " [0].drLastName ="Cope" [0].dateOfConsultation="1964-05-12" [0].treatment="Alzen" [0].treatment_efficiency=false
		[1].id=153 [1].disease="migraine" .....
		.....
2002	.....	.....

Fig. 10: Cassandra (datastax) database

```
{
"idP": 100,
"FirstName": "Mary",
"LastName": "Jane",
"Gender": "F",
"DateOfBirth": new Date ('Jun 23, 1970'),
"PlaceOfBirth": "Omaha, Nebraska",
"PlaceOfResidence": "Omaha, Nebraska",
"Consultations": [
        {"disease": "diabetes", "drFirstName": "John", "drLastName": "Bradley",
        "DateOfConsultation": new Date ('Feb 23, 1992'), "treatment": "insulin",
        "treatment_efficiency": true},
        .....
            ]
}
```

Fig 11: MongoDB (mlab) database

The extraction and loading code in the target collection:

```
import nosqlapi.*;

public class test {

    public static void main(String[] args) {

        RedisHandler h1= new RedisHandler();
        h1.extract("hospital1" , "Patient");

        CassandraHandler h2= new CassandraHandler();
        h2.extract("hospital2" , "Patient");

        MongoDBHandler h3= new RedisHandler();
        h1.extract("hospital3" , "Patient");

        h1.serialize("hospital", "Patients");
        h2.serialize("hospital", "Patients");
        h3.serialize("hospital", "Patients");
    }
}
```

Listing 1: A test JAVA code

We can notice that with our API, access to several NoSQL databases has become so easy. After the execution of this program, we will have the following result:

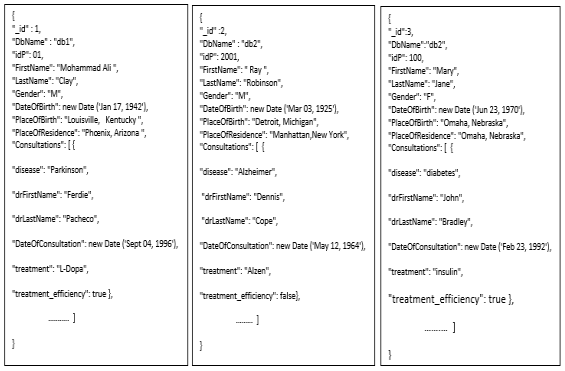

Fig. 12: Data after serialization

Now, we can do analysis, for example, there are several possibilities to know the number of people affected by Parkinson's disease:

First alternative (MongoDB Aggregation Pipeline):

```
hospital.Patients.aggregate(
   [
      {
         $match: {
            disease: {
               $eq: "Parkinson"
            }
         }
      },
      {
         $count: "patients_count"
      }
   ]
)
```

Second alternative (MapReduce):

```
var mapFunction1 = function() {
                       emit(this.disease, 1);
                   };
var reduceFunction1 = function(disease, values) {
                          return Array.sum(values);
                       };
db.orders.mapReduce(
                     mapFunction1,
                     reduceFunction1,
                     { out: "map_reduce_example" }
                   )
```

Third alternative (Apache Spark):

```
val rdd = MongoSpark.load(sc)
val filteredRdd = rdd.filter(doc => doc.getString("disease") == "Parkinson")
println(filteredRdd.count)
```

8. Conclusion

In this article, we introduced a new approach for real time big data analytics on heterogeneous NoSQL dabases. Our approach is based on the following principle: to develop an application that queries or analyzes data distributed in several NoSQL databases (data sources). We begin with a phase of integration of these data into another target NoSQL database to prepare them for the query. Since data sources can have different data models (key-store, document store or columnar store). This phase requires the transformation of data to the model of the target database before the integration of these data. Thus, we have developed an API for the integration of distributed data in a single data warehouse. Finally, we applied our approach to the healthcare case study to show that our approach facilitates the analysis and exploration of data stored in multiple data sources.

References

[1] Martin, F.: Polyglot persistence (November 2011)
[2] M. Stonebraker, “Stonebraker on nosql and enterprises,” Commun. ACM, vol. 54, no. 8, pp. 10–11, 2011.
[3] Intel It Center , “Big Data in the Cloud: Converging Technologies, How to Create Competitive Advantage Using Cloud-Based Big Data Analytics”,2015 .
[4] NIST Big Data Interoperability Framework: Volume 1, Definitions ,2015, http://dx.doi.org/10.6028/NIST.SP.1500-1
[5] L. Cabibbo, “Ondm: an object-nosql datastore mapper,” Faculty of Engineering, Roma Tre University. Retrieved June15th, 2013.
[6] F. Bugiotti, L. Cabibbo, P. Atzeni, R. Torlone ,” A Logical Approach to NoSQL Databases,” .
[7] Pollack, M., et al.: Spring Data. Volume 1. O'Reilly Media (October 2012)
[8] Paolo Atzeni, Francesca Bugiotti, and Luca Rossi , “Uniform access to non-relational database systems: the SOS platform”,(2012), Advanced Information Systems Engineering.
[9] R Sellami, S Bhiri, B Defude : ODBAPI: a unified REST API for relational and NoSQL data stores , Big Data (BigData Congress), 2014 IEEE International Congress on, 653-660
[10] T. Haselmann, G. Thies, and G. Vossen, “Looking into a rest based universal api for database-as-a-service systems,” in 12thIEEE Conference on Commerce and Enterprise Computing,CEC 2010, Shanghai, China, November 10-12, 2010,pp. 17–24.
[11] D. Chatziantoniou :ODMC: Towards an Interoperable Big Data Universe. ACM SIGMOD Record (2013).
[12] H. M. L. Dharmasiri , M. D. J. S. Goonetillake : federated approach on heterogeneous NoSQL data stores, International Conference on Advances in ICT for Emerging Regions (ICTer), 2013
[13] H. Lim, Y. Han, S. Babu: How to fit when no one size fits. In: CIDR 2013, sixthBiennial Conference on Innovative Data Systems Research, Asilomar, CA, USA,January 6-9, Online Proceedings. (2013)
[14] J. Castrejon, G. Vargas-Solar,C. Collet, R. Lozano,” Model-Driven Cloud Data Storage,”.
[15] O Hajoui ; R Dehbi ; M Talea ; A Bakhouyi ; Z Ibn Batouta , A comparative analysis of different approaches for big data interoperability ,Third International Conference on Systems of Collaboration (SysCo),2016
[16] M ZAHARIA et al., Apache Spark:A Unified Engine for Big Data Processing”, COMMUNICATIONS OF THE ACM NOVEMBER 2016, VOL. 59 NO. 11.
[17] https://spark.apache.org/, 2018
[18] https://hadoop.apache.org/