Ingénierie des Systèmes d'Information
Vol. 28, No. 2, April, 2023, pp. 315-325
Journal homepage: http://iieta.org/journals/isi

A Schema Integration Approach for Big Data Analysis

Souad Amghar*, Safae Cherdal, Salma Mouline

LRIT, Faculty of Sciences, Mohammed V University in Rabat, Rabat 10080, Morocco

Corresponding Author Email: souad_amghar@um5.ac.ma

https://doi.org/10.18280/isi.280207

Received: 22 February 2023
Accepted: 2 April 2023

Keywords:
data integration, NoSQL systems, schema integration, schema matching

ABSTRACT

A huge volume of data is analyzed by organizations to understand their clients and improve their services. In many cases, these data are stored separately in different database systems and need to be integrated before being used in analysis tools or prediction applications. One of the main tasks of data integration process is the definition of the global schema. Defining a global schema in the context of NoSQL systems is a demanding task since it necessitates dealing with a variety of issues, including the lack of local schemas, data model heterogeneity, and semantic heterogeneity. To address these challenges, this work aims to automatically define the global schema of a set of databases stored in heterogeneous NoSQL systems. The main contributions of this work are presented in three phases: (1) Schema extraction where we define the local schemas using a unified representation. (2) Schema matching in which we propose a hybrid approach to find matching attributes between the local schemas. (3) Schema integration where we define the global schema using the schema matching results. A Covid-19 use case as well as other benchmarks are presented in this paper to evaluate the results of the proposed approach and illustrate its effectiveness.

1. INTRODUCTION

Context

Today, practically every organization has transformed into a data-driven organization, which means that they are using a method to gather data from various sources and analyze them to derive valuable information [1].

In addition to their large volume, these data are heterogenous and most of the time unstructured. Therefore, they cannot be managed by relational systems. The adoption of NoSQL systems is the result of all these specificities.

NoSQL which stands for Not Only SQL is used to describe non-relational, distributed, and scalable systems.

These systems offer high availability, simple scalability, and support for numerous data structures, making them excellent for managing massive volumes of data [2].

Unlike relational systems that use a unified data model representation and query language, NoSQL systems use different approaches and query languages [3], and they are grouped into four categories which are document-oriented, column-oriented, key-value, and graph-oriented [4].

To analyze data stored in different categories of NoSQL systems we need to integrate them. Data integration is the problem of combining data stored in many database systems to provide a unified view of data through a common representation called the global schema [5].

Problems

The definition of the global schema is one of the main tasks in the design of a data integration system. Defining a global schema in the context of NoSQL systems is a demanding task since it requires addressing several challenges:

- Lack of local schemas: Most NoSQL systems are schemaless which means that the database does not follow a predefined schema. This means that each record may have a distinct schema.
- Data model heterogeneity: Each NoSQL system uses a different data model to store data. For instance, data in document-oriented systems are stored as JSON or BSON documents, whereas data in graph-oriented systems are stored as graphs.
- Semantic heterogeneity: Data are not stored to be integrated. They are stored independently for various purposes. As a result, the names and terminology used to represent data are different.

Contributions

Because of the lack of works that address the previously mentioned challenges, this work aims to automatically define the global schema of a set of databases stored in heterogeneous NoSQL systems. The main contributions of this work are presented in three phases:

- Schema extraction phase where we define the local schemas using a unified representation.
- Schema matching phase in which we propose a hybrid approach to specify matching attributes between the local schemas.
- Schema integration phase where we propose a methodology to define the global schema using the results of the schema matching.

Our proposition aims to provide data analysts with a unified view of data stored in various NoSQL systems. This unified view can be used in big data analysis to query data across several sources as well as to import data from multiple sources into one source.

Paper structure

The rest of this paper is organized as follows: Section 2 presents some of the related works. Section 3 gives in detail the phases of the proposed approach. We provide in section 4 an evaluation of our proposed approach using a Covid-19 use

case and a set of benchmarks. Conclusions and future works are given in section 5.

2. RELATED WORKS

According to a review of 794 articles presented by Guo et al. [6], there is a lack of data integration solutions in the works that use data analysis. Most data analysis applications use a single-sourced methodology that might have left out crucial indicators and produced biased algorithms. According to this review, the full potential of data analysis can be realized by integrating heterogeneous data, which will also increase accuracy and decrease bias.

One of the few works that performed heterogeneous data integration for data analysis is KG-COVID-19 [7]. It offers a method for integrating heterogeneous data from many sources to produce knowledge graphs. KG-COVID-19 enables users to make complex queries about pertinent biological entities and to utilize machine learning analysis for predictions. Although this solution offers a quick way to combine fresh data and information from many sources, it is dedicated to the biomedical research community since it integrates data about drugs and gene expression. Moreover, this work does not use NoSQL systems.

Ramadhan et al. [8] propose a semi-automatic schema integration approach composed of two phases which are schema matching and schema mapping. In the schema matching phase, a similarity score is defined for the different attributes of local schemas by comparing schema instances as well as attribute names and datatypes using a string-based similarity measure and a WordNet based semantic similarity measure. The schema mapping phase generates the global schema by merging the local schemas concepts that have matching attributes. This work provides an interesting hybrid approach for schema matching. However, schema integration is defined by merging schemas that have at least one match which may lead to merging concepts that are not similar.

Madhavan et al. [9] is a schema matching system that uses a hybrid approach to identify matches between schema items based on their names, datatypes, constraints, and schema structure. Input schemas are encoded as graphs where nodes represent schema elements and edges represent the hierarchical relation between them. This approach uses string-based and structural similarity measures to create final matching by selecting pairs of schema components that have a similarity coefficient greater than the threshold. Cupid proposes an interesting solution for schema matching based on many evaluation experiments. However, it tends to rely largely on predefined domain synonyms and abbreviations which are not always easy to identify.

The work presented in research [10] is a schema matching solution that uses two techniques of the instance-based approach. The first technique uses regular expressions to compare numeric and mixed data. The second technique uses a semantic similarity measure. The combination of these two techniques enables the system to achieve a high F-measure. However, this solution is based on instances only which makes it unsuitable for small datasets.

Radwan et al. [11] propose a top-k ranking algorithm that generates a set of integrated schemas. This approach calculates the similarity of schemas' concepts based on the similarity of their attributes by using Hausdorff Distance measure. The concepts with a similarity score higher than a predefined threshold are either merged or connected with a 'has' relationship. The main goal of this work is to reduce the manual effort needed to define the global schema. However, this solution is limited to relational and XML systems.

We provide in Table 1 a summary of the previously presented related works where we present their main contribution and the database systems used.

Table 1. Related works summary

Work	Contributions	Database systems
[7]	Graph Merging	YAML Files
[8]	Schema Matching Schema Integration	Relational, NoSQL, and HDFS Systems
[9]	Schema Matching	Relational and XML Systems
[10]	Schema Matching	Not Specified
[11]	Schema Integration	Relational and XML Systems

By analyzing the previously presented related works, as shown in Table 1, we can retrieve the following needs:

- Few works include schema extraction in the global schema definition process. The vast majority begin with predefined local schemas. In our context, we assume that we do not have any information about the integrated databases, and we need to extract the local schemas.
- Most existing works use one schema matching technique. Many studies, however, show that using hybrid approaches is the key to overcoming the limitations of each technique [12]. Moreover, all matching techniques can generate incorrect results. However, none of the existing solutions provide a post-processing step to identify and remove misleading findings.
- The majority of existing schema integration solutions are either manual or semi-automatic. The automatic approaches are proposed in the context of relational or XML systems. However, we need new methodologies to automatically define the global schema in the context of NoSQL systems.

To address the previously presented limitations, we present in this paper a data integration approach to combine heterogeneous NoSQL schemas in a unified view called the global schema. This unified view can be used in big data analysis to import data from several sources into one source as well as to query data across numerous sources.

3. THE PROPOSED APPROACH

To automatically define the global schema of a set of databases stored in heterogeneous NoSQL systems, we need to go through three phases: schema extraction, schema matching, and schema integration. Each phase takes place in several steps. Figure 1 represents the architecture of the approach which is composed of three phases: schema extraction, schema matching and schema integration.

Schema Extraction: In this phase, we automatically define the local schemas by extracting general information such as the name and datatype of the attributes. We also propose a unified representation of the local schemas to alleviate the data model heterogeneity of NoSQL systems.

Schema matching: We specify in this phase, matching attributes in the different local schemas by combining datatype-based, semantic-based, string-based, and instance-based techniques. We also suggest a post-processing step that helps in detecting and deleting incorrect matches.

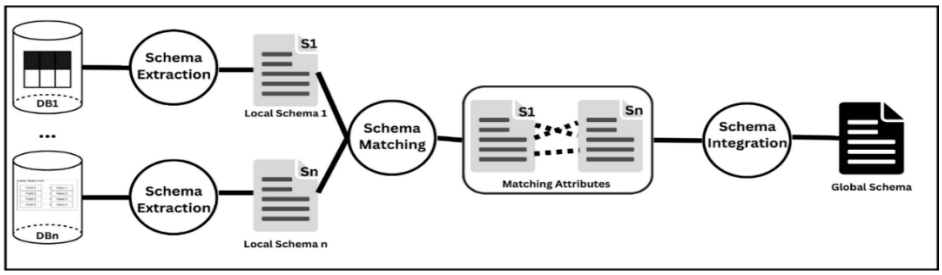

Figure 1. The approach architecture

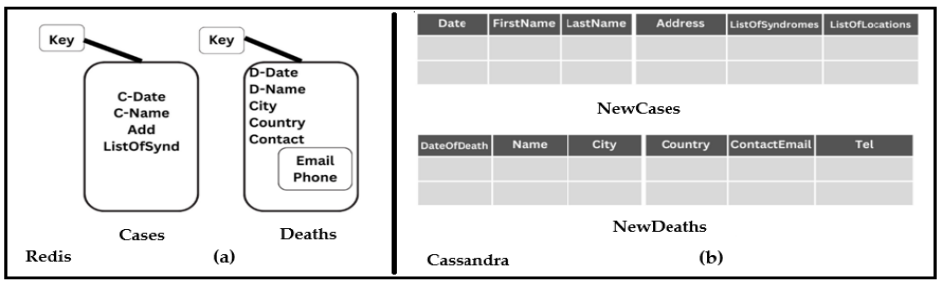

Figure 2. (a) DailyReport database stored in Redis; (b) Report database stored in Cassandra

Schema Integration: In this phase, we exploit the results of schema matching to identify matching concepts that are merged to define the global schema.

To illustrate the different phases of the proposed approach, we use an example related to healthcare data where we define the global schema of two databases namely **DailyReport** (a) and **Report** (b) stored respectively in Redis and Cassandra as shown in Figure 2. The two databases contain data about cases and deaths related to Covid-19 [5].

3.1 Schema extraction

Extracting schema from NoSQL systems is a difficult task because of many challenges. First, a lot of NoSQL systems do not include any tools for extracting schema. Second, most NoSQL systems are schema-less, which means they do not require the database schema to be defined before storing data. As a result, we may end up having many schemas for the same database. Third, extracting schema from a system that is document-oriented differs from extracting schema from a system that is key-value, column-oriented, or graph-oriented as each type uses a different data model.

To represent the different schemas in a unified way, we need to clearly define the meaning of the following items: a database, a concept, and an attribute:

- A database is a set of data that is saved in a database management system. For instance, the Report database stored in Cassandra (Figure 2).
- A concept is a database component made up of related data. For instance, the Report database is composed of two concepts: **NewCases** and **NewDeaths**.
- An attribute is a concept field describing its characteristics and properties. For example, **NewCases** concept has attributes such as Date, FirstName, and LastName.

Given the heterogeneity of NoSQL systems types, the previously presented items can have different significations. In Table 2, we present the equivalents of database, concept, and attribute in five different NoSQL systems representing the four existing categories.

Table 2. The equivalents of database, concept, and attribute in five NoSQL systems

NoSQL system	NoSQL system type	Database	Concept	Attribute
Redis	Key-value	A Redis database	A key-value pair or a Hash	A key-value pair
Cassandra	Column-oriented	A keyspace	A table	Column or a Column family
MongoDB	Document-oriented	A MongoDB database	A collection	A document's field
Couchbase	Key-value and Document- oriented	A Couchbase bucket	A collection	A document's field
Neo4j	Graph-oriented	A graph	A node	A node's or relationship's property

Table 3. Schema extraction tools for NoSQL systems

NoSQL system	Concepts names	Attributes names	Attribute types
Redis	A script using SCAN or HSCAN	A script using SCAN or HSCAN	A script using TYPE
Cassandra	"system schema" command	"system schema" command	"system schema" command
MongoDB	"extract schema" command	"extract schema" command	"extract schema" command
Couchbase	A script using SELECT *	A script using SELECT *	A script using SELECT *
Neo4j	Call apoc.meta.schema()	Call apoc.meta.schema()	Call apoc.meta.schema()

In our approach, the main items that need to be extracted from the database are concept names, attribute names, and attribute types. We extract this information using the system's extraction predefined tool, such as the 'system schema' command in Cassandra. For the systems that do not provide a schema extraction tool, we define a Python script that goes through the database and extracts the various concepts as well as the names and types of attributes.

We present in Table 3 the implementation details of the schema extraction phase for five NoSQL systems representing the four categories. Nevertheless, this solution can be extended to other NoSQL systems.

After extracting the main components of the local schema, we define a unified representation based on JSON [13]. The reason for choosing JSON is its flexibility and self-describing nature in addition to its ease of use in many programming languages. We consider that the local schema is a JSON object that is composed of three key-value pairs: Database-System-Name, Database-Name, and Concepts. The Concept contains the concept's name as well as the related attributes. Each attribute is presented using its name, type, and if present, a set of sub-attributes as presented in Figure 3.

```
Local schema ::= {Database_System_Name Database_Name Concepts}
Database_System_Name ::= "Database_System_Name": Name,
Database_Name ::= "Database_Name": Name,
Concepts ::= "Concepts":[ Concept (, Concept)*]
Concept ::= { "Concept_Name" : Name, Attributes }
Attributes::= "Attributes" :[ Attribute (, Attribute)* ]
Attribute ::= {"Attribute_Name": Name, "Attribute_Type": typeName} |
{"Attribute Name": Name, "Attribute Type": "object", Attributes}
typeName := "string" | "integer" | "number" | "boolean" | "date" |
            "null" | "array"
Name ::= string
```

Figure 3. The unified representation syntax of local schemas

In order to illustrate the schema extraction phase, we return to the example illustrated in Figure 2 of two databases namely **DailyReport** (a) and **Report** (b) stored respectively in Redis and Cassandra. Redis database is stored as two hashes (Cases and Deaths). However, Cassandra database is composed of two tables (NewCases and NewDeaths).

The extracted schemas of Cassandra and Redis databases are presented in Figure 4 using the unified representation.

Every table becomes a concept for Casandra, and every hash becomes a concept for Redis.

The schema extraction phase enables the generation of the local schemas with a unified representation which alleviates the data model heterogeneity of NoSQL databases. In the next phase, which is schema matching, we address the semantic heterogeneity issue by specifying matching attributes between local schemas.

3.2 Schema matching

The goal of schema matching is to solve the semantic heterogeneity issue of NoSQL databases. In this phase, we aim to specify matching attributes between local schemas.

The main problem of schema matching is that data are stored independently with different reasoning. For this reason, finding matching attributes is difficult.

There are various schema matching techniques, each of which serves a particular purpose and has its limits. For instance, the semantic-based technique which uses the semantic similarity of names cannot provide good results for attributes whose names do not convey any semantic information such as ListofSynd (Example of Figure 2). These matches can only be found using the string-based or the instance-based technique. To overcome the different limitations of the different techniques, our work is based on a hybrid schema matching approach in which we use datatype-based, semantic-based, string-based, and instance-based matchers which are the most used techniques in schema matching [14].

To illustrate the schema matching approach, we use the illustration example shown in Figure 2 in which we identify matching attributes between Cassandra and Redis local schemas. Only to make it easier to represent matching attributes, the two local schemas are shown in a hierarchical representation, as seen in Figure 5.

In this example, the concepts Cases and NewCases share five true matches, and the concepts Deaths and NewDeaths share six additional true matches. Dashed lines serve as a representation of the true matches.

The first step of the schema matching phase uses the datatype-based technique where we generate a list of matching candidates of two schemas which are the attributes with the same type. This step prevents false matches and reduces the number of elements that must be processed by the other three matchers.

```
{"Database System": "Redis",
"Database Name": "DailyReport",
"Concepts": [
{"Concept Name": "Cases",
"Attributes": [
{"Attribute Name": "C–Date", "Attribute Type": "date"},
{"Attribute Name": "C–Name", "Attribute Type": "String"},
{"Attribute Name": "Add", "Attribute Type": "String" },
...]},
{"Concept Name": "NewDeaths",
{"Attribute Name": "D–Death", "Attribute Type": "date"},
{"Attribute Name": "Name", "Attribute Type": "String"},
...
{"Attribute Name": "Contact", "Attribute Type": "Object",
"Attributes":[
{"Attribute Name": "email", "Attribute Type": "String"},
{"Attribute Name": "phone", "Attribute Type": "String"} ]}]}]}
```

```
{"Database System": "Cassandra",
"Database Name": "Report",
"Concepts": [
{"Concept Name": "NewCases",
"Attributes": [
{"Attribute Name": "Date", "Attribute Type": "date"},
{"Attribute Name": "FirstName", "Attribute Type": "String"},
{"Attribute Name": "LastName", "Attribute Type": "String" },
{"Attribute Name": "address", "Attribute Type": "String" },
...]},
{"Concept Name": "NewDeaths",
{"Attribute Name": "DateOfDeath", "Attribute Type": "date"},
{"Attribute Name": "Name", "Attribute Type": "String"},
{"Attribute Name": "City", "Attribute Type": "String" },
{"Attribute Name": "Country", "Attribute Type": "String" },
...]}
]}
```

Figure 4. Local schemas of Redis and Cassandra databases presented in the unified representation

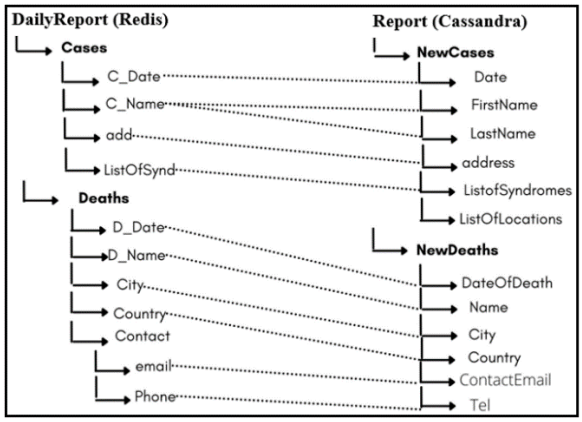

Figure 5. True matching attributes of Redis and Cassandra databases

The three matchers algorithms take separately as input the results of the datatype-based matcher which we refer to as matching candidates. Similarly to the literature, our algorithms produce false matches (i.e. False positives). For this reason, we define post-processing that detects and deletes false positives of each matcher. After applying the post-processing, the union of the three matchers results constitutes the schema matching phase's results.

We describe in detail the algorithms of the three proposed matchers and the post-processing in the following sub-sections.

3.2.1 Semantic-based matcher

The semantic-based technique enables the specification of matching attributes based on the semantic similarity of their names. There are two categories of semantic similarity measures. The first category is corpus-based. It derives the similarity score of two texts using large corpora. The second category is knowledge-based, it identifies the degree of similarity between words using information generated from semantic networks such as WordNet [15]. In our approach, we are interested in knowledge-based similarity measures because corpus-based measures have a statistical background and do not take into consideration the actual meaning of words which may produce a lot of false results. Based on a set of experiments, where we compare many knowledge-based similarity measures, we have decided to use Resnik [9] and Leacock & Chodorow [9] measures in our semantic-based matcher algorithm.

Algorithm 1. Semantic-based matcher

```
1   function Semantic-based-Matcher (Candidates)
2   // Candidates is a list of possible matching attributes generated by the type-based matcher
3   // Candidates = {a1 = b1, a2 = b2, .., an = bn}
4   Initialize a List of SemanticMatches;
5   for i from 1 to size(Candidates) do
6       ai ← clean(ai );
7       bi ← clean(bi );
8   if ResnikSimilarity(ai,bi)>=0.6 or LeacockChodorowSimilarity ai,bi)>=0.6 then
9           Add ai = bi to SemanticMatches
10      end
11  end
11  return SemanticMatches;
```

We define the semantic-based matcher as described in Algorithm 1. The algorithm takes as input a set of candidates produced by the datatype-based matcher and returns a set of matching attributes. For each candidate, the algorithm first cleans the names of the attributes to get rid of punctuation, special characters, and uppercase. Then, it calculates the similarity of each candidate using Resnik and Leacock&Chodorow similarity measures and selects those where at least one of the similarity scores is greater or equal to 0.6 (lines 8-9). The threshold has been chosen based on many experiments.

In this algorithm, the similarity scores produced by Resnik and Leacock&Chodorow measures are normalized to provide values between 0 and 1.

To illustrate the semantic-based matcher, we return to the example of Figure 5. As presented in Table 4, the matcher finds eight correct matches. However, it also returns eight false matches because of the semantic similarity of these words. These false matches are detected and deleted in the post-processing step.

Table 4. Semantic-based matcher results

True Positives	False Positives
C-Date=Date;	C-Date=DateOfDeath;
C-Name=FirstName;	C-Name=Name;
C-Name=LastName;	D-Date=Date;
D-Date=DateOfDeath;	D-Name=FirstName;
D-Name=Name;	D-Name=LastName;
City=City;	City=Country;
Country=Country;	Country=City;
email=ContactEmail	Contact=ContactEmail

The semantic-based matcher finds eight out of eleven matches which demonstrates the necessity of the other matchers.

3.2.2 String-based matcher

The string-based technique enables the identification of matching attributes based on the similarity of their strings. Many string-based similarity measures are proposed in the literature. Some of them are character-based measures which make them suitable for comparing simple words. Another category of string-based measures is token-based. This category recognizes similarities between two groups of words. It is suitable for the comparison of long texts [9].

In our context, attributes' names are, in general, composed of one word or parts of many words. For this reason, character-

based measures are the most suitable in our case. Consequently, we choose to use Jaro-Winkler measure [9] as it is the most recommended measure in this category.

Similarly, to Algorithm 1, we define the string-based matcher algorithm that takes as input the same set of candidates produced by the type-based matcher and returns a set of matches as shown in Algorithm 2. It enables the identification of matching attributes based on the similarity of their characters using Jaro-Winkler measure. In this algorithm, the threshold of 0.77 has been chosen based on many experiments.

Algorithm 2. String-based matching

```
1   function String-based-Matcher (Candidates)
2   // Candidates is a list of possible matching
attributes generated by the type-based matcher
3   // Candidates = {a1=b1, a2=b2, .., an=bn}
4   Initialize a List of StringBasedMatches;
5   for i from 1 to size (Candidates) do
6       ai ← clean(ai);
7       bi ← clean(bi);
8       if jaroWinklerSimilarity (ai, bi) >= 0.77 then
9           Add ai = bi to StringBasedMatches
10      end
11  end
12  return StringBasedMatches;
```

We return to the example of Figure 5 to illustrate the string-based matcher. As shown in Table 5, the matcher finds eight correct matches, one of which is not found by the semantic-based matcher. However, it also returns four false matches.

Table 5. String-based matcher results

True positives	False positives
C-Date=Date; add=address; ListOfSynd=ListofSyndromes; D-Date=DateOfDeath; D-Name=Name; City=City; Country=Country; email=ContactEmail	D-Name=Name; ListOfSynd= ListOfLocations; D-Date=Date; Contact=ContactEmail;

Even though we use the string-based matcher, not all matches are found. As a result, we define the instance-based macher as discussed in the subsection that follows.

3.2.3 Instance-based matcher

Instance-based matcher considers attributes similar if their instances are similar [16]. This technique is valuable in cases where attributes are named using different words or even different languages. It enables the specification of matching attributes that can not be generated by string-based and semantic-based techniques.

The instance-based matcher is presented in Algorithm 3. It takes the same set of candidates produced by the datatype-based matcher as input and returns a set of matches. Since this matcher uses instances of the attributes, we need to access the databases. After establishing the database connexion, the algorithm first generates attributes' instances for every matching candidate (lines 9-10). Then, it calculates the number of values in common for each candidate (lines 11-16). After that, it asserts that the percentage of equal values is more than 0.6 in both databases. Like the other two matchers, the threshold of 0.6 is selected after conducting numerous experiments.

Algorithm 3. Instance-based matching

```
1  function Instance-based-Matcher (Candidates)
2  // Candidates is a list of possible matching attributes
generated by the type-based matcher
3  // Candidates = {a1 = b1, a2 = b2, .., an = bn}
4  Connect(Database1);
5  Connect(Database2);
6  Initialize a List of InstanceBasedMatches;
7  for i from 1 to size(Candidates) do
8      Set elementInCommun to zero;
9      queryResultsList1 ← Select ai from Database1;
10     queryResultList2 ← Select bi from Database2;
11     for j from 1 to size(queryResultsList1) do
12         for j from 1 to size(queryResultsList2) do
13             if element1=element2 then
14                 elementInCommun ← elementInCommun+1
15             end
16         end
17     if elementInCommun/size(queryResultsList1) >= 0.6 AND
18         elementInCommun/size(queryResultsList2) >= 0.6 then
19         ADD ai = bi to InstanceBasedMatches;
20     end
21 end
22     return InstanceBasedMatches;
```

For our example (Figure 5), the instance-based matcher returns five correct matches which are: C-Date=Date; D-Date=DateOfDeath; City=City; Country=Country; Phone=Tel, and it does not return any false positive. The Phone=Tel match is not found by the other matchers which shows the effectiveness of our hybrid approach.

We provide in the following, a post-processing step to automatically identify and delete misleading results. In our work, the proposed post-processing only concerns the semantic-based and string-based matchers since they generate more incorrect results than the instance-based matcher.

3.2.4 Post-processing

The main goal of the post-processing step is to distinguish between true and false matches. It is based on a set of rules that validate the results of semantic and string-based matchers separately.

Let M be a set of matches produced by each matcher as follow: M={a1=b1, a2=b2, .., an=bm}. M is the union of M_s which is a set of 1:1 matches that are single matching attributes (Ex. {C-Date=Date; City=City}) and M_m that is a set of 1:n matches that contains multi-matching attributes (Ex. {C-name=FirstName; C-name=LastName}).

The post-processing concerns only 1:n matching attributes which are more likely to contain false results. We can have false 1:1 matching results, however, during our experiments, we found that some of these false matches are removed in the schema integration phase. For this reason, we restricted the post-processing to the 1:n matching attributes.

The post-processing algorithm takes as input an attribute 'a' from both local schemas as well as a set of its 1:n matches and

removes the false matches from this set as provided in Algorithm 4.

The algorithm is performed separately on the results of the semantic-based matcher and the string-based matcher using the threshold and the similarity measures provided in each matcher.

The post-processing algorithm is composed of three major parts each of which deals with a different category of false matches. We explain each part using the example of Figure 5.

Algorithm 4. Post-processing algorithm

```
1  function Post-processing (a, Mm)
2  //Mm is a list of matches of attribute a
3  // Mm = {a = a1, a = a2.., a = an}
4      if ∃i∈{1, 2, .., n}/IsSimilar (ConceptName(a),ConceptName(ai )) then
5    for j from 1 to n do
6      if ¬IsSimilar (ConceptName(a), ConceptName(ai)) then
7          Delete {a = aj} from Mm;
8      end
9      if ∃i∈ {1, 2, .., n}/Similarity (a, ai) == 1 then
10         for j from 1 to n do
11             if Similarity (a, ai) < 1 then
12                 Delete {a = aj} from Mm;
13         end
14     else
15         amax = MaxSimilarity(Mm);
16         for j from 1 to n / aj ! = amax do
17     if isNotSimilar(amax,aj) OR isHierarchical(amax,aj) then
18                 Delete {a = aj} from Mm;
19           end
20         end
21     end
22    end
23     return Mm;
```

The first part of our post-processing algorithm (Algorithm 4 lines 4-8) addresses the case where an attribute 'a' of a concept 'C' is matched with many attributes 'a_i' of different concepts 'C_i'. In this case, we calculate the semantic similarity between the name of concept 'C' and the name of each concept 'C_i' using Resnik and Leacock&Chodorow measures as presented in Algorithm 1 of the semantic-based matcher. If the name of concept 'C' is similar to at least one of the concepts 'C_i', then we delete all matches where the name of concepts are not similar.

For instance, in the semantic-based matcher's results of the example of Figure 5, the attribute C-Name is matched with FirstName and LastName, as well as with Name. All of These results are semantically correct because there is a semantic similarity between them. However, in this example, C-Name should not be matched with Name since the concept's name Cases is semantically similar to NewCases but not similar to NewDeaths.

For our example, this step helps in removing the five false matches from the semantic-based matcher: C-Date=DateOfDeath, C-Name=Name, D-Date=Date, D-Name=FirstName, and D-Name=LastName. And two false matches from the string-based matcher: C-Name=Name and D-Date=Date.

The second part of the post-processing (Algorithm 4 lines 9-13) deals with the case where there is a similarity of 1 between two attributes. In general, identical attributes are not supposed to have 1:n matches. For this reason, we remove all the other matches where the similarity score is less than 1.

This case is illustrated by the example of Country and City. Since there is a semantic similarity between the words Country and City, the attribute Country is matched with Country and with City. However, since the similarity between Country and Country equals 1, we delete the match Country=City.

In this step, we remove two false matches from the semantic-based matcher: Country=City and City=Country.

In the last part of the post-processing (Algorithm 4 lines 14-19) we deal with matches where the similarity scores between attributes are less than 1.

We notice that in a true 1:n match between an attribute 'a' and a set of attributes 'a_i' there is often a high similarity between 'a_i'. Based on this idea, we first identify attribute 'a_{max}' that has the highest similarity score with attribute 'a'. Then, we remove the matches in which the attribute 'a_i' is not similar to 'a_{max}'.

For instance, C-Name is matched with FirstName and LastName which is a true match. However, ListOfSynd is matched with ListofSyndromes and ListofLocations and should be matched only with ListofSyndromes. The difference between the match of C-Name and the match of ListOfSynd is that there is a similarity between FirstName and LastName, but no similarity between ListofSyndromes and ListofSyndromes. For this reason, we only keep the match ListofSynd=ListofSyndromes since it has the highest similarity score (a_{max}). As a result, in this step, we remove the false match: ListOfSynd=ListofLocations from the string-based matcher results.

Another case is also handled in this part using the condition isHierarchical (a_{max}, a_j). In this case, if there is a hierarchy between 'a_i' and 'a_{max}' we only keep the match a=a_{max}. For instance, Contact and Email are hierarchical, and they are both matched to ContactEmail in the result of the semantic-based and string-based matchers. Thanks to this condition we delete the false match Contact=ContactEmail.

The post-processing of our approach helps in improving the outcomes of the various matchers by identifying and removing misleading findings. After applying the post-processing, the final result of each matcher is the union of all results of the post-processing step and the set of 1:1 matches (M_s).

The final result of the schema matching phase is the union of the three matchers results after applying the post-processing. The matching attributes are used to define the global schema in the schema integration phase as provided in the following section.

3.3 Schema integration

The schema integration phase enables the generation of the global schema of a set of local schemas. This phase exploits the results of schema matching to decide which concepts are merged in the global schema.

The concepts that have many matching attributes are more likely to be similar. Based on this idea, we define the similarity score of each two concepts as the ratio of their matching attributes over the total number of attributes.

Let, for a concept, NMA be the Number of Matching Attributes and NA be the Number of Attributes, we first define a Directed Concept Similarity score (DCS) for each concept as defined in Eq. (1):

$$DCS_{concept1} = \frac{NMA}{NA} \quad (1)$$

Then, we calculate the Concept Similarity score (CS) of two concepts as the average of their Directed Concept Similarity scores as defined in Eq. (2):

$$CS_{concept1,concept1} = \frac{DCS_{concept1} + DCS_{concept2}}{2} \quad (2)$$

The global schema is generated by merging the concepts that have a Concept Similarity score higher than a threshold. The choice of the threshold is based on a series of experiments in which the best results were obtained when the threshold is set to 0.6.

To illustrate the effectiveness of the schema integration phase, we return to the example of Figure 5. In this example, the concepts Cases and NewCases are supposed to be merged, and Deaths should be merged with NewDeaths.

Table 6. CS scores of Covid-19 databases

Concept1	Concept2	Positives	CS score
Cases	NewCases	5	$\frac{1.25+0.83}{2} = 1.04$
Cases	NewDeaths	0	$\frac{0+0}{2} = 0$
Deaths	NewCases	0	$\frac{0+0}{2} = 0$
Deaths	NewDeaths	6	$\frac{0.86+1}{2} = 0.93$


```
{"Database System": "",
"Database Name": "GlobalSchema",
"Concepts": [
{"Concept Name": "Cases",
"Attributes": [
{"Attribute Name": "C-Date", "Attribute Type": "date"},
{"Attribute Name": "C-Name", "Attribute Type": "String"},
{"Attribute Name": "Add", "Attribute Type": "String" },
{"Attribute Name": "ListOfSynd", "Attribute Type": "String" }
{"Attribute Name": "ListOfLocations", "Attribute Type": "String"
  }]},
{"Concept Name": "NewDeaths",
{"Attribute Name": "D-Death", "Attribute Type": "date"},
{"Attribute Name": "Name", "Attribute Type": "String"},
{"Attribute Name": "City", "Attribute Type": "String"},
{"Attribute Name": "Country", "Attribute Type": "String"},
{"Attribute Name": "Contact", "Attribute Type": "Object",
"Attributes":[
{"Attribute Name": "email", "Attribute Type": "String"},
{"Attribute Name": "phone", "Attribute Type": "String"} ]}]}]}
```


Figure 6. The Global schema of Redis and Cassandra databases

The results shown in Table 6 demonstrate a considerable difference in the Concept Similarity scores (CS) of Cases and NewCases as well as of Deaths and NewDeaths when compared to the other two scores. Therefore, the global schema is defined using the unified representation and composed of two concepts: Cases which is the merge of Cases (Redis) and NewCases (Cassandra), and Deaths which is the merge of Deaths and NewDeaths. As shown in Figure 6, each concept of the global schema contains concept attributes of the first local schema (Cases) in addition to the unmatched attributes of the concept of the second local schema (NewCases).

Moreover, the schema integration phase enables also to delete some false matches. For instance, if the CS score of two concepts C1 and C2 is less than 0.6, the false matches (1:1 or 1:n matches) between these concepts are eliminated because the two concepts are not merged. Thus, the schema integration phase enables the definition of the global schema and consequently reduces the number of false matches.

The three phases of the proposed approach enable the automatic definition of the global schema of databases stored in two NoSQL systems. We provide in the following section, the implementation of our approach using a real Covid-19 use case and a set of benchmarks.

4. IMPLEMENTATION AND RESULTS EVALUATION

To prove the effectiveness of the proposed work, we first evaluate our approach using a real use case that uses Covid-19 related data. Then, we use a set of existing benchmarks.

4.1 Use case

We propose, in this paper, a use case based on Covid-19 related data to illustrate the effectiveness of our approach. This use case evaluates the results of the proposed approach and illustrates some possible situations that can arise in the definition of the global schema.

4.1.1 Databases

We use real Covid-19 databases which are SarsCov2Data [17, 18], CovidData [17, 19], and CovidGlobalData [17] as shown in Figure 7.

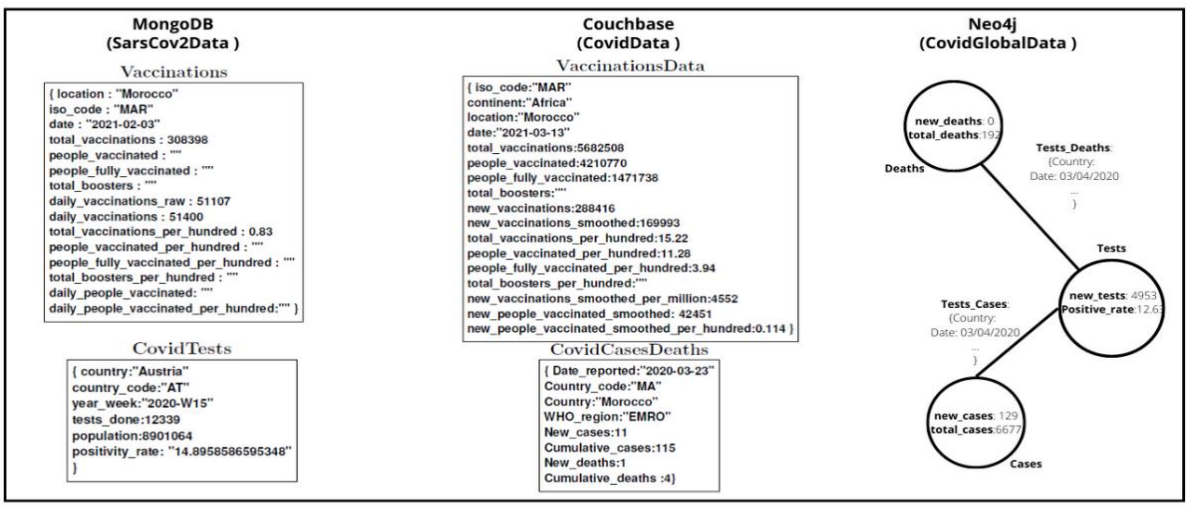

Figure 7. Local schemas of MongoDB, Couchbase, and Neo4j

The database SarsCov2Data is stored in MongoDB as two collections of BSON documents: Vaccinations (86.936 records) and CovidTests (13.000 records). The database CovidData is stored in Couchbase using two collections: CovidCasesDeaths (187.180 records) and VaccinationsData (166.554 records). However, the database CovidGlobalData is stored in Neo4j as three nodes: Cases (166.554 records), Deaths (166.554 records), and Tests (166.554 records).

In order to obtain the global schema of these three databases, it is necessary to go through the three phases of the proposed approach: Schema extraction, schema matching, and schema integration.

4.1.2 Schema extraction

The initial phase in the approach provided in this work is to extract the local schemas of various NoSQL systems as shown in Figure 7.

For MongoDB, we use Mongo-inspector, a Python package that allows us to extract the schema of a MongoDB database. It provides a list of the names and types of characteristics of the database collections [20]. In Neo4j, apoc.meta.schema() allows the extraction of information about nodes, relations, and properties [21]. For Couchbase, we define a python script that scans the database and extracts attribute names and types because no schema extraction tool is provided by this system [22].

The unified representations of the extracted schemas are defined similarly to the example in Figure 4.

4.1.3 Schema matching

The second phase of the proposed approach aims at finding matching attributes between the three local schemas previously generated by the schema extraction phase. We evaluate the schema-matching results using Precision, Recall, and F-measure metric [23].

According to the evaluation values presented in Table 7, the proposed approach achieves interesting results by finding all matching attributes. The false positives generated by our approach can be explained by the fact that the attribute names in the same schema are close to each other, for instance, daily_vaccinations_raw and daily_vaccinations in SarsCov2Data database.

4.1.4 Schema integration

The last phase of the approach is schema integration where we automatically define the global schema.

The first step of this phase is to compute the concept similarity scores (CS) as shown in Table 8. In this phase, we merge concepts that have a concept similarity score (CS) greater than 0.6.

CovidCasesDeaths is matched with two concepts Cases and Deaths. As a result, the concept CovidCasesDeaths is merged with Cases and Deaths. Consequently, as presented in Figure 8, the global schema is composed of three concepts Vaccinations, CovidTest, and CovidCasesDeaths which is a correct global schema.

This use case shows that in concrete situations, our approach allows to unify the representation of local schemas and the definition of the global schema in order to query and analyze heterogeneous databases which is not proposed by existing approaches.

Table 7. Results evaluation of schema matching of Covid-19 use case

Local schema 1	Local schema 2	True matches	Positives	True positives	Precision	Recall	F-measure
Vaccinations	VaccinationsData	15	19	15	0.79	1	0.88
CovidCasesDeaths	Cases	5	6	5	0.83	1	0.91
CovidCasesDeaths	Deaths	5	6	5	0.83	1	0.91
CovidTests	Tests	5	5	4	0.8	0.8	0.8

Table 8. Concept similarity scores of Covid-19 use case

Local Schema 1	Local Schema 2	Positives	CS Score
Vaccinations	VaccinationsData	19	**1.15**
CovidTest	Tests	5	**0.73**
CovidCasesDeaths	Cases	6	**0.75**
CovidCasesDeaths	**Deaths**	**6**	**0.75**


```
{"Database System": "",
"Database Name": "GlobalSchema",
"Concepts": [
{"Concept Name": "Vaccinations",
"Attributes": [
{"Attribute Name": "location", "Attribute Type": "string"},
{"Attribute Name": "iso_code", "Attribute Type": "String"},
{"Attribute Name": "date", "Attribute Type": "date" },
...]},
{"Concept Name": "CovidTest",
{"Attribute Name": "country", "Attribute Type": "string"},
{"Attribute Name": "country_code", "Attribute Type": "String"},
...]}
{"Concept Name": "CovidCasesDeaths",
{"Attribute Name": "Date_reported", "Attribute Type": "date"},
{"Attribute Name": "Country_code", "Attribute Type": "String"},
...]}
]}
```


Figure 8. The global schema of the Covid-19 use case

4.2 Benchmark tests

In addition to the use case, we use two different benchmarks to evaluate the schema matching phase. Due to the lack of benchmarks that can be used to validate the definition of the global schema in the context of NoSQL systems, we use two XML-based benchmarks to evaluate our approach.

The first benchmark is XBenchMatch [24] which has a set of matching attributes defined in different scenarios.

The second benchmark is the Purchase-Order benchmark [9] which contains a set of matching attributes between various Purchase and Order schemas.

Because there is no data to evaluate our instance-based matcher in these two benchmarks, we use just the string-based and semantic-based matchers.

Table 9. The evaluation metrics of the schema matching phase for the benchmarks

Benchmark	Positive	True Positive	Precision	Recall	F-measure
XBenchMatch	6	6	1	1	1
Purchase-Order benchmark	30	26	0.87	0.93	0.89

Using only the string-based and the semantic-based matchers in addition to the post-processing, we have the same matching attributes as the XBenchMatch benchmark.

For Purchase-Order benchmark, we identify 30 matches using our schema matching approach, 26 of which are true matches. The four false positives are caused either by the semantic similarities of words. The false negatives, which are the matches that should be identified but are not, have high similarity scores but fall short of the threshold.

Due to the lack of space, we only present in Table 9 the evaluation metrics of the schema matching phase for the used benchmarks.

Using these two benchmarks in addition to the use case, we prove the effectiveness of our approach in schema extraction, schema matching, and schema integration.

5. CONCLUSIONS AND FUTURE WORKS

In this paper, we addressed the problem of defining the global schema for data analysis in the context of NoSQL systems.

As presented in related works (Section 2), the majority of existing schema integration solutions are either manual or semi-automatic. The automatic approaches are proposed in the context of relational or XML systems. Our approach, however, enabled the automatic definition the global schema of data stored in various NoSQL systems by providing solutions for schema extraction, schema matching, and schema integration.

In schema extraction, we proposed a method to generate the local schemas and present them in a unified representation which solves the data model heterogeneity problem of NoSQL systems.

We provided a hybrid schema matching approach that uses type-based, semantic-based, string-based, and instance-based matchers in addition to the post-processing that gave interesting results according to the evaluation metrics.

We also proposed a data integration methodology that defined the global schema by merging the concepts that have a high similarity score. The benchmarks as well as the Covid-19 use case presented in this paper illustrated the effectiveness of our approach through various scenarios.

The three phases of the proposed approach allowed for the alleviation of challenges related to NoSQL systems including the lack of local schemas, the data models heterogeneity and the semantic heterogeneity of data.

For future work, we are working on creating a user interface that makes it easier to use our solution in data analysis process and allows user intervention to build a global schema that meets data analysis requirements. Moreover, we are working on a query processing solution that uses the global schema definition approach to query data across several sources.

REFERENCES

[1] El Aissi, M.E.M., Benjelloun, S., Lakhrissi, Y., El Haj Ben Ali, S. (2022). Big data enabling fish farming data-driven strategy. Ingénierie des Systèmes d'Information, 27(6): 949-956. https://doi.org/10.18280/isi.270611

[2] Amghar, S., Cherdal, S., Mouline, S. (2018). Which NoSQL database for IoT applications. In 2018 International Conference on Selected Topics in Mobile and Wireless Networking (MoWNeT), Tangier, Morocco, 131-137. https://doi.org/10.1109/MoWNet.2018.8428922

[3] Amghar, S., Cherdal, S., Mouline, S. (2022). Storing, preprocessing and analyzing tweets: Finding the suitable noSQL system. International Journal of Computers and Applications, 44(6): 586-595. https://doi.org/10.1080/1206212X.2020.1846946

[4] Amghar, S., Cherdal, S., Mouline, S. (2019). Data integration and noSQL systems: A state of the art. In Proceedings of the 4th International Conference on Big Data and Internet of Things, pp. 1-6. https://doi.org/10.1145/3372938.3372954

[5] Olivé, A. (2018). A universal ontology-based approach to data integration. Enterprise Modelling and Information Systems Architectures (EMISAJ), 13: 110-119. https://doi.org/10.18417/emisa.si.hcm.10

[6] Guo, Y., Zhang, Y., Lyu, T., Prosperi, M., Wang, F., Xu, H., Bian, J. (2021). The application of artificial intelligence and data integration in COVID-19 studies: A scoping review. Journal of the American Medical Informatics Association, 28(9): 2050-2067. https://doi.org/10.1093/jamia/ocab098

[7] Reese, J.T., Unni, D., Callahan, T.J., Cappelletti, L., Ravanmehr, V., Carbon, S., Mungall, C.J. (2021). KG-COVID-19: A framework to produce customized knowledge graphs for COVID-19 response. Patterns, 2(1): 100155. https://doi.org/10.1016/j.patter.2020.100155

[8] Ramadhan, H., Indikawati, F.I., Kwon, J., Koo, B. (2020). MusQ: A Multi-store query system for IoT data using a datalog-like language. IEEE Access, 8: 58032-58056. https://doi.org/10.1109/ACCESS.2020.2982472

[9] Madhavan, J., Bernstein, P.A., Rahm, E. (2001). Generic schema matching with cupid. In VLDB, pp. 49-58.

[10] Mahdi, A.M., Tiun, S. (2014). Utilizing wordnet and regular expressions for instance-based schema matching. Research Journal of Applied Sciences, Engineering and Technology, 8(4): 460-470. https://doi.org/10.19026/rjaset.8.994

[11] Radwan, A., Popa, L., Stanoi, I.R., Younis, A. (2009). Top-k generation of integrated schemas based on directed and weighted correspondences. In Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data, pp. 641-654. https://doi.org/10.1145/1559845.1559913

[12] Doan, A., Halevy, A., Ives, Z. (2012). Principles of data integration. Elsevier. ISBN: 9780123914798

[13] Pezoa, F., Reutter, J.L., Suarez, F., Ugarte, M., Vrgoč, D. (2016). Foundations of JSON schema. In Proceedings of the 25th International Conference on World Wide Web, pp. 263-273. https://doi.org/10.1145/2872427.2883029

[14] Bellahsene, Z., Bonifati, A., Duchateau, F., Velegrakis,

Y. (2011). On evaluating schema matching and mapping pp. 253-291. Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-642-16518-4_9
[15] Miller, G.A. (1995). WordNet: A lexical database for English. Communications of the ACM, 38(11): 39-41. https://doi.org/10.1145/219717.219748
[16] Rahm, E., Bernstein, P.A. (2001). A survey of approaches to automatic schema matching. The VLDB Journal, 10: 334-350. https://doi.org/10.1007/s007780100057
[17] Hannah Ritchie, E.M., Rodés-Guirao, L., Appel, C., Giattino, C., Ortiz-Ospina, E., Hasell, J., Roser, M. (2020). Coronavirus pandemic (COVID-19). Our World in Data. Available: https://ourworldindata.org/coronavirus.
[18] https://www.ecdc.europa.eu/en/publications-data/ covid-19-testing, accessed on Feb. 20, 2023.
[19] https://covid19.who.int/info, accessed on Feb. 20, 2023.
[20] https://pypi.org/project/mongo-inspector/, accessed on Feb. 20, 2023.
[21] https://neo4j.com/labs/apoc/4.1/overview/apoc.meta/apoc.meta.schema/, accessed on Feb. 20, 2023.
[22] https://docs.couchbase.com/server/current/n1ql/n1ql-language-reference/infer.html, accessed on Feb. 20, 2023.
[23] Dhar, J., Jodder, A.K. (2020). An effective recommendation system to forecast the best educational program using machine learning classification algorithms. Ingénierie des Systèmes d'Information, 25(5): 559-568. https://doi.org/10.18280/isi.250502
[24] Duchateau, F., Bellahsene, Z., Hunt, E. (2007). XBenchMatch: A benchmark for XML schema matching tools. The VLDB Journal, 1: 1318-1321.