2018 IEEE International Conference on Information Reuse and Integration for Data Science

An Approach for Schema Extraction of JSON and Extended JSON Document Collections

Angelo Augusto Frozza
Catatarinense Federal Institute
Camboriú – SC, Brazil
angelo.frozza@ifc.edu.br

Ronaldo dos Santos Mello, Felipe de Souza da Costa
Federal University of Santa Catarina
Florianópolis – SC, Brazil
r.mello@ufsc.br, feekosta@outlook.com

***Abstract*—JSON documents are raising as a common format for data representation due to the increasing popularity of NoSQL document-oriented databases. One of the reasons for such a popularity is their ability to handle large volumes of data at the absence of an explicit data schema. However, schema information is sometimes essential for applications during data retrieval, integration and analysis tasks, for example. Given this context, this paper presents an approach that extracts a schema from a JSON or Extended JSON document collection stored in a NoSQL document-oriented database or other document repository. Aggregation operations are considered in order to obtain a schema for each distinct structure in the collection, and a hierarchical data structure is proposed to group these schemas in order to generate a global schema in JSON Schema format. Experiments conducted on actual datasets, like DBPedia and Foursquare, demonstrate that the accuracy of the generated schemas is equivalent or even superior than related work.**

Keywords: NoSQL; JSON; Extended JSON; Schema Extraction; JSON Schema; Document-Oriented Database.

I. Introduction

The *Big Data* market explosion has led large companies to demand databases (DB) that are able to store and process large volumes of data effectively. In this context, traditional relational DB present several limitations as they prioritize strong consistency for read and write operations instead of high availability and horizontal expansion to better deal with increasing data volumes [1].

One family of new DB systems that have emerged to cope with these relational DB limitations is called NoSQL DB [2]. Basically, they avoid the overhead with the traditional ACID properties for transaction management, providing, instead, an eventual data consistency, a strong availability and elasticity capabilities. A common feature of NoSQL DB is that they are *schemaless*, i.e., they allow the storage of data without prior knowledge of their structure [3]. Document-oriented DB, one of the NoSQL database categories, for example, store and retrieve documents with simple and complex attributes mainly in JSON (*JavaScript Object Notation*) or Extended JSON[1] formats ([1], [4]), and documents do not necessarily share a common structure. In fact, schemaless DB avoid the problem of records with several attributes without a value when they are not uniform, allowing each record to contain only what is necessary.

However, the lack of information regarding the schema makes difficult to perform data integration processes, as well as several data processing tasks, such as data retrieval, validation and analysis [5]. Nevertheless, the absence of an explicit schema does not mean the total absence of a schema, since there is usually an implicit schema in the application code that access the database [6]. In the NoSQL DB context, to be aware of a data collection schema is very important during an application development. For example, several applications dealing with geographic data, such as *Foursquare*, retrieve data in JSON format, but do not define a schema for them, being difficult for users to query such data because they are unaware of the documents structure.

Based on this motivation, this paper presents an approach that aims to generate a single schema from a collection of JSON or Extended JSON documents. The generated schema is defined in the *JSON Schema* recommendation, which is establishing as a standard for specifying schemas of JSON documents [7]. The proposed approach, called *JSON Schema Discovery*, intends to mainly aid applications that need to be aware of NoSQL document-oriented DB schema for several purposes, like NoSQL schema and data integration. We give emphasis here to NoSQL document-oriented DB due to their popularity, but, in fact, our approach is able to produce a JSON Schema for any JSON document collection that it receives as input.

Different from related work, our approach provides the generation of a schema not only for a set of JSON documents, but also for a set of Extended JSON documents, and defines a hierarchical data structure that hold several metadata information useful for the schema generation. This hierarchical structure is manipulated by a *Model Driven Engineering (MDE)*-based process, which is a suitable software development technique for dealing with data model transformations [8]. Besides, our solution is available as a web tool that allows the generation, persistence, view and download of schemas in JSON Schema format.

Experiments to verify the quality of the schemas generated

[1] https://goo.gl/1QA7EY

978-1-5386-2659-7/18/$31.00 ©2018 IEEE
DOI 10.1109/IRI.2018.00060

using our approach were performed on controlled databases as well as databases used by related work. Some schemas generated by our approach were more accurate due to the identification of the possible data types for each field of the schema. It highlights that it is promising.

The rest of this paper is organized as follows. Section 2 gives a brief overview of JSON, Extended JSON and JSON Schema. Section 3 presents the related work. Section 4 details our approach. Section 5 presents an evaluation of our approach and section 6 is dedicated to the conclusion.

II. JSON, Extended JSON and JSON Schema

JSON[2] is a standard format used for complex and heterogeneous data exchange [9]. A unit of data represented in JSON is a *document*. Figure 1 shows an example of JSON document. Its structure comprises the following concepts:

1) ***Object***: a disordered set of key-value pairs. It is limited by '{' (open braces) and '}' (close braces), and each key-value pair is separated by ',' (comma). A key is a *string* content followed by ':' (colon) and the value is the third concept detailed in the following. One example in Figure 1 is the object *javascriptWithScopeType*, which is composed of two key-value pairs.
2) ***Array***: an orderly collection of values. It is limited by '[' (open brackets) and ']' (close brackets) and the values are separated by ', ' (comma). Examples are *arrayMultiValues* and *arrayOfNumber* in Figure 1.
3) ***Value***: A JSON value can be a primitive type value (string, numeric, boolean), a structured type (object, array) or *null*. On using objects and arrays as values it is possible to define JSON documents with nested structures. The values of the object and arrays of Figure 1 mentioned above are examples.

MongoDB[3], the most popular NoSQL document-oriented database, stores JSON documents in a binary-encoded format called BSON (*Binary JSON*)[4], which additionally supports data types that are not part of the original JSON specification, such as *Date*, *Timestamp*, *Binary*, *ObjectID*, *RegExp*, *Long*, *Undefined*, *DBRef*, *Code*, *MinKey* and *MaxKey*. This is known as Extended JSON. Figure 1 shows, in fact, a document with Extended JSON data types.

JSON Schema is an attempt to provide a general purpose schema language for JSON, which allows users to restrict the structure of JSON documents, including required objects and the data types allowed for object attributes or array values [7]. JSON Schema is in the process of becoming a recommendation by the IETF - *Internet Engineering Task Force*, currently published as *draft-07*[5].

[2]https://www.json.org
[3]https://www.mongodb.com
[4]http://bsonspec.org/
[5]http://json-schema.org

Figure 1. Extended JSON Document (*Doc-3*)

```
{
    "_id": ObjectId("50319491fe4dce143835c552"),
    "binaryType": BinData(2, "dGVzdGFuZG8="),
    "dateAsISO": Date("2014-01-31T22:26:33.000Z"),
    "timestampType": Timestamp(1421006159, 4),
    "dbRefType": DBRef("some_collection",
            "50319491fe4dce143835c552"),
    "undefinedType":undefined,
    "longType": NumberLong("9223372036854775807"),
    "booleanType": false,
    "nullType": null,
    "regexType": /\/^S\//m,
    "stringType": "some_value",
    "objectType": { "property": "some_value" },
    "arrayMultiValues": [ 48, "50",
        {"property": "teste"},  [1,"some_value"],
        true, BinData(2, "dGVzdGFuZG8=")   ],
    "arrayOfNumber": [ 1, 2 ],
    "javascriptType": { "$code":
        "var a = function(param){console.log(param);};"},
    "javascriptWithScopeType": {
        "$code": "int x = y", "$scope": { "y": 1 } },
    "maxKeyType": { "$maxKey": 1 },
    "minKeyType": { "$minKey": 4 },
}
```

Figure 5 shows the first part of the JSON Schema specification for the JSON document of Figure 1. This part (*definitions* section) allows the definition of properties (e.g., *"ObjectID"* in Figure 5) that may be reused in the second part (*"properties"* section - Figure 6) of the schema specification (e.g., *"#/definitions/ObjectID"* in Figure 6). Each item in *"properties"* defines an attribute. The *"additionalProperties": false* property indicates that no additional attributes are allowed in a multivalued attribute, whereas *"required": ["property"]* says that the cited attributes are required.

Constraints can be applied to an attribute by adding specific clauses in the schema. For example, the clause *"type"* is used to restrict an attribute to a JSON type, like *array* and *string*. The clause *"anyOf "* says that an attribute may hold one of the listed data types, like *"anyOf ": ["string", "number "]*. For example, the *arrayMultiValues* attribute in Figure 6 defines an array that can have elements with the data type *number*, *string*, *object*, *arrayMultiValues*, *boolean* or a reference (*$ref*) to a property definition *Binary*.

III. Related Work

Klettke et al. [10] propose a process for extracting schemas from *MongoDB*. It considers a data structure called *Structure Identification Graph (SG)* that summarizes the structural information of all analysed documents. At the end, a JSON Schema is derived from the SG. It is checked which attributes are required or optional as well as their data types. For a property to be considered mandatory, it must appear in all JSON documents. The schema also specifies if an attribute supports more than one data type.

The work of Ruiz et al. [6] proposes a reverse engineering process for extracting versions of document-oriented NoSQL

DB schemas. Initially, a *MapReduce*[6]-based procedure is performed to extract a collection of JSON objects containing an object for each version (structure) of a real-world entity stored in the database. Next, the collection of JSON objects is injected into a JSON model that conforms to a JSON metamodel, defined using MDE techniques. Finally, a reverse engineering process receives the JSON model and outputs a model that conforms to a NoSQL schema metamodel.

In the work of Wang et al. [11], a framework is proposed for schema management of NoSQL document-oriented DB. One of its tasks is to identify distinct schemas in a data collection, grouping the equivalent ones in a proprietary hierarchical data structure called *eSiBU-Tree* (*encoded Schema in BUcket Tree*). Only attribute names are considered to assemble the record schema, not referring to data types.

The work of Izquierdo et al. [12] generates the schema of a set of JSON documents obtained from a web service. It executes an MDE-based process in which new obtained JSON documents contribute to enrich the generated schema. Initially, JSON documents are obtained from multiple calls to a web service. Next, a JSON schema represented in the ECORE model[7] is generated for each JSON document based on a JSON metamodel. At the end, the ECORE schemas are merged to obtain a global schema.

The work of Wischenbart et al. [5] proposes an MDE-based approach for extracting user profile schemas from social networks (Facebook, Google+ and LinkedIn). First, JSON documents are extracted from social networks through their corresponding APIs. Next, multiple schemas are inferred from the data previously extracted. These schemas are expressed in JSON Schema and are further merged to produce a global schema in the ECORE model.

More recently, Baazizi et al. [13] introduced an algorithm to infer a schema from JSON datasets. It consists of two main steps: (*i*) input JSON values are processed by a *Map* function to infer a simple data type for each value; (*ii*) the output is processed by a *Reduce* function, which iteratively makes a fusion of the inferred types, identifying mandatory, optional or repeated elements in the data types. The final schema is described in a language based on JSON Schema.

A comparison of the related work is shown in Table I. It includes our approach (*JSON Schema Discovery*). Some approaches consider specific NoSQL document-oriented DB (e.g., *MongoDB*) as data sources of JSON documents, while others accomplish a process (API execution or social network extraction) to generate the input JSON data. Our approach initially focused on *MongoDB* for experimental evaluation purposes. Even so, our extraction process is able to deal with any JSON document collection received as input. Besides, we are the only approach that supports schema extraction from Extended JSON documents.

Table I also shows that two different approaches are followed to perform the schema extraction: hierarchical structures to summarize schema information, or MDE-based processes. Our work combines both approaches and aggregation operations to produce the output JSON schema.

Intermediate data models (or hierarchical data structures) are considered by some approaches (including ours) to maintain extracted data schema information from JSON documents before the generation of the output schema. It avoids the management of an intermediate (and incomplete) JSON Schema specification during the extraction process, giving that JSON schema has a complex structure. Additionally, it would be expected that the output schema be generated in JSON Schema, as it is raising as a standard. Only some approaches (including ours) consider JSON Schema.

With respect to the implementation of the proposed approaches, most of them presents only high level algorithms or framework architectures without providing the source codes for reproducibility purposes. Only the works of Ruiz el al. [6] and Izquierdo et al. [12] give access to their source codes. Different from all of them, our work is the only one that offers a *SaaS* (*Software-as-a-Service*) solution accessed through any web browser.

IV. *JSON Schema Discovery*

JSON Schema Discovery is an approach for schema extraction of JSON (or Extended JSON) document collections. Our extraction process initially traverses all JSON documents and analyses their properties to identify the schema of each document. Next, it unifies these document schemas and generates a single schema, in JSON Schema format, that represents the collection of documents as a whole. To summarize the structural information of all raw schemas, a tree data structure is defined to maintain their properties: nested objects, arrays, primitive JSON data types or Extended JSON.

The process is organized into four steps detailed as follows: *(i)* Document raw schema generation; *(ii)* Grouping of document raw schemas; *(iii)* Unification of document raw schemas; *(iv)* JSON Schema generation. As stated before, our *SaaS* tool focuses on schema extraction from *MongoDB* collections because it is the most used NoSQL document-oriented DB. However, our approach is able to deal with any JSON or Extended JSON document collection.

A. Raw Schema Generation

The first step of our approach is the generation of the schema of each JSON document in a collection (called *raw schema*), and its storage in a temporary *MongoDB* collection. The raw schema contains the same structure of the original JSON document with respect to attributes, nested objects and arrays. Each primitive value in the document is replaced in the raw schema by its JSON data type (e.g., *string* and

[6]https://docs.mongodb.com/manual/core/map-reduce/

[7]https://www.eclipse.org/modeling/emf/

Table I
RELATED WORK COMPARISON

	Klettke et al. [10]	**Ruiz et al.** [6]	**Wang et al.** [11]	**Izquierdo et al.** [12]	**Wischenbart et al.** [5]	**Baazizi et al.** [13]	***JSON Schema Discovery***
Data Source	MongoDB dataset	MongoDB, CouchDB, HBase	JSON datasets	Web service APIs	Social network APIs	JSON datasets	MongoDB dataset
Approach	Hierarchical Summarization	MDE	Hierarchical Summarization	MDE	MDE	Hierarchical Summarization	Hierarchical Summarization and MDE
Input Format	JSON	JSON	JSON	JSON	JSON	JSON	JSON and Extended JSON
Intermediate Model	*SG*	JSON Model			JSON Schema		RSUS
Output Format	JSON Schema	NoSQL DB Schema	*eSiBu-Tree*	ECORE Schema	ECORE Schema	Proprietary Schema	JSON Schema
Implementation	Algorithm	Algorithm	Framework	Eclipse Plugin	Algorithm	Algorithm	SaaS

number), including the data types of the Extended JSON (e.g., *date* and *objectId*). Algorithm 1 presents the raw schema extraction process. It receives a collection of JSON documents as input and outputs a collection of raw schemas.

The *BuildRawSchema()* function receives a value as input and extracts the raw schema from the value. In the first time it is invoked, it receives the document key, and it recursively traverses the document structure obtaining the key of each attribute and extracting the schema of its value. If the value data type is a primitive or Extended JSON type, the algorithm returns the type name. Otherwise, a recursion is applied to the structured values (*object* or *array* data types).

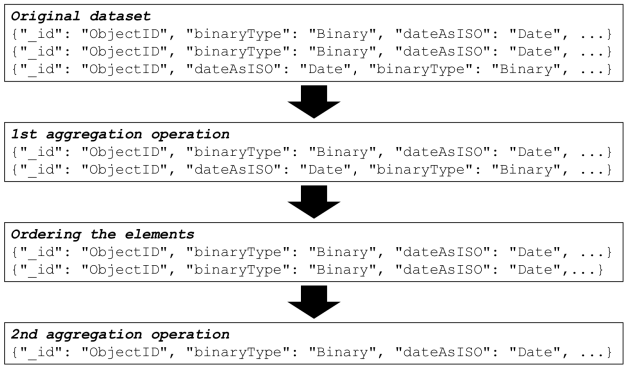
Algorithm 1: Raw Schema Generation

```
1  function Parser(collection);
   Input  : collection of JSON documents
   Output: rawSchemas of JSON documents
2  begin
3      rawSchemas ← ∅;
4      for document ∈ collection do
5          documentRawSchema ← {};
6          for key ∈ keys(document) do
7              value ← document[key];
8              documentRawSchema[key] ←
                 BuildRawSchema(value);
9          end
10         add documentRawSchema to rawSchemas;
11     end
12     return rawSchemas;
13 end
```

B. Grouping of Raw Schemas

In this step, two aggregation operations are applied over raw schemas to extract a collection of unique JSON objects, i.e., the minimum number of objects required to perform the raw schemas unification process. Figure 2 demonstrates this step, which is explained in the sequence. We adopt aggregation operations from NoSQL DB. They present a better performance, if compared to *MapReduce* functions implemented in the application code because they are native (and optimized) operations able to deal with a large data volume.

Figure 2. Grouping of Raw Schemas



The first aggregation operation works on the resulting collection from the first step in order to group the documents that have the same raw schema. The result is stored in a temporary collection that is accessed by the second aggregation operation.

Before executing the second aggregation operation, the attributes of the raw schemas obtained from the first aggregation operation are disposed in alphabetical order. The purpose of this ordering is to reduce the number of documents with different raw schemas, since the elements of a JSON document do not need to follow the same order as defined in the schema. After the ordering is done, the second aggregation operation is executed to remove duplicate raw schemas, which results in the smallest possible number of documents with different raw schemas.

C. Unification of Raw Schemas

A tree-based data structure, called ***R**aw **S**chema **U**nified **S**tructure* (RSUS), is defined in this step to store information about the hierarchical structure of the raw schemas. The considered information are the attributes of the objects and their data types, the elements of the arrays and their data types, the path of the document root to an attribute or any

item, and the number of attribute occurrences based on the path and the data type. These information are essential to the generation of the final schema, including the definition of the data type(s) of each attribute and if it is mandatory or not. It is important to observe here that we do not perform semantic analysis of attribute names. We only check whether a given attribute exists in the RSUS. Compared to related work, RSUS provides more metadata information than other hierarchical data structures, like the support to Extended JSON data types.

Figure 3. RSUS Data Structure

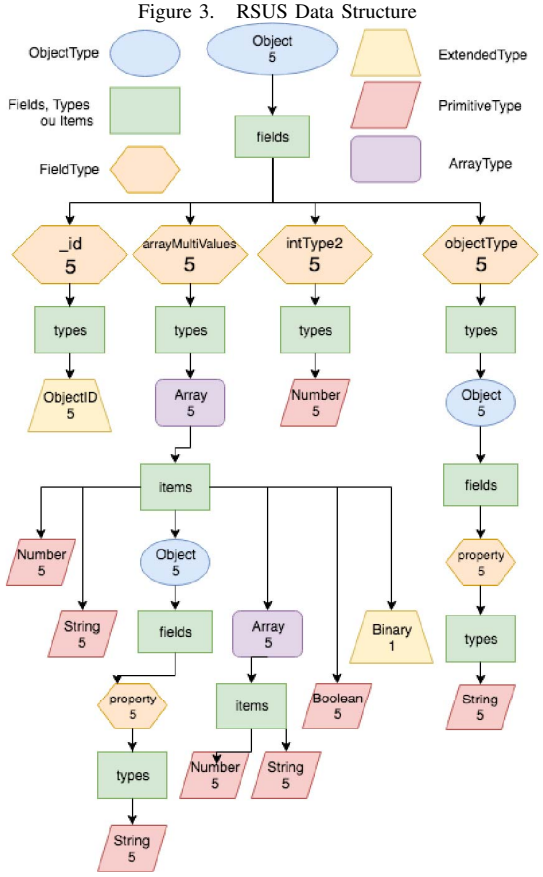


Figure 3 shows a graphical representation of an RSUS with some information about the schema of the JSON document of Figure 1. The RSUS holds five attribute types:

- ***FieldType***: it represents a document attribute composed of: *(i) "name"*: attribute name; *(ii) "path"*: path from document root to the attribute. *(iii) "count"*: number of attribute occurrences in the documents; *(iv) "types"*: an array containing the data types associated to the attribute in the documents: *"primitiveType"*, *"extendedType"*, *"objectType"* ou *"arrayType"*.
- ***PrimitiveType***: it represents a primitive JSON data type composed of: *(i) "name"*: one of the following data type names: *"String"*, *"Number"*, *"Boolean"* ou *"Null"*; *(ii) "path"*: path from the document root to the data type; *(iii) "count"*: number of occurrences of the data type in attributes of the documents.
- ***ExtendedType***: it holds the same attributes of the previous type for representing an extended JSON type. The attribute *name* holds one of the possible Extended JSON types as value.
- ***ObjectType***: it represents an object composed of the attributes *path* and *count* with the same structure of the two previous data types, and two additional attributes: *(i) "name"*: attribute with the value *"Object"*; *(ii) "fields"*: an array containing the attributes that occur for this object in the documents. Each array element is an RSUS attribute of the type *"FieldType"*.
- ***ArrayType***: it represents an array composed of the attributes *path* and *count*, and the other following attributes: *(i) "name"*: attribute with the value *"Array"*; *(i) "items"*: an array containing the elements that occur in this array for the extracted documents. An element is a property whose type may be *"ObjectType"*, *"PrimitiveType"*, *"ExtendedType"* or *"ArrayType"*.

The RSUS is built from the extracted raw schemas. The elements of each raw schema are traversed in pre-order and, for each element, the addition or extension of an attribute in the RSUS is performed. If the element represents an object attribute, then an addition or extension of a *FieldType* attribute in the RSUS is accomplished. If the element represents a JSON or Extended JSON type, then an addition or extension of a RSUS *Type* attribute (*Primitive, Extended, Object or Array*) in RSUS is performed. Finally, if it represents an array element, then an addition or extension of an item in RSUS is accomplished. Any addition or extension takes into account the path from the root of the raw schema to the property being analysed.

Besides, the *count* field in RSUS (the numbers in some nodes of Figure 3) has two purposes: (*i*) to determine whether a schema element is required (the number of the element must be equal to the number of the RSUS root); (*ii*) to determine whether an element has more than one data type (the data type count is less than the element's count).

D. JSON Schema Generation

Once the raw schemas are summarized at RSUS, we execute the last step of our extraction process through an algorithm that transforms the RSUS structure into a valid JSON Schema (Algorithm 2). The general idea of this step is to generate a JSON Schema for each attribute in the RSUS respecting the hierarchical structure of the raw schema.

Basically, each RSUS attribute is mapped to its respective definition in JSON Schema. The *getSchemaFromValue()* function checks for the type of the RSUS attribute and generates the JSON Schema for this type.

Some examples of outputs generated by the execution of *JSON Schema Discovery* steps are given in the next section.

Algorithm 2: RSUS → JSON Schema Mapping

```
function getFieldsSchema(fields);
  Input  : fields
  Output: properties
begin
    properties ← {};
    for field ∈ fields do
        properties[field.name] ←
          getSchemaFromValue(field);
    end
    return properties;
end
```

V. JSON Schema Discovery Evaluation

We considered three experiments to evaluate the quality of the generated schemas by *JSON Schema Discovery* . Different collections of JSON documents stored in *MongoDB* were used on each of them.

A. Quality of JSON document mapping for JSON Schema

In this evaluation, we verify the correctness and completeness of the JSON → JSON Schema mappings. To do so, five JSON documents with several data types and heterogeneous structures were created, as shown in Table II.

Table II
INPUT JSON DOCUMENTS

Document	Content
Doc-1	Base document with major JSON and JSON extended data types.
Doc-2	Doc-1 with changes in the data values.
Doc-3	JSON document with values representing all Extended JSON data types, and an array containing items with different data types.
Doc-4	Similar to Doc-3, but some attributes and items of the arrays were deleted. It was created to represent optional attributes.
Doc-5	Similar to Doc-4, but the attributes are in different order.

Doc-1 e *Doc-2* have the same structure and the same data type in the attributes. The only difference is on the attribute values. In this case, the first aggregation operation of the Grouping step considered these documents as having the same raw schema, as expected.

Doc-3 (Figure 1) it is the most detailed document of this experiment, having all types supported by Extended JSON and an array including items of different data types. Figure 4 shows the raw schema of this document obtained by the first step of the *JSON Schema Discovery* process.

Doc-4 has a document similar to that of Figure 1 (*Doc-3*). However, it does not have an item of type *"Binary"* in the *"arrayMultiValues"*, and the *"arrayOfObject"* property has only one object type, whereas *Doc-3* has two ones. This document allows you to test the existence of optional and required attributes.

Doc-5 is a document similar to *Doc-4*. The only difference between them is the order of the properties. In this case, as expected again, the second aggregation operation of the Grouping step, in which the raw schema properties are first ordered, considered these two documents as having the same raw schema.

Figure 4. Raw Schema (*Doc-3*)

```
{
  "_id": "ObjectID",
  "binaryType": "Binary",
  "dateAsISO": "Date",
  "timestampType": "Timestamp",
  "dbRefType": "DBRef",
  "undefinedType": "Undefined",
  "longType": "Long",
  "booleanType": "Boolean",
  "nullType": "Null",
  "regexType": "RegExp",
  "stringType": "String",
  "objectType": { "property": "String" },
  "arrayMultiValues": [
    "Number", "String",
    { "property": "String" },
    [ "Number", "String" ],
    "Boolean", "Binary"    ],
  "arrayOfNumber": [ "Number" ],
  "javascriptType": "Code",
  "javascriptWithScopeType": "Code",
  "maxKeyType": "MaxKey",
  "minKeyType": "MinKey",
}
```

In summary, the first aggregation operation of the second step of *JSON Schema Discovery* returned four documents, as *Doc-1* and *Doc-2* have the same raw schema. After sorting the raw schema properties of the four documents, *Doc-4* and *Doc-5* now have the same schema, so the second aggregate operation returns three documents.

In the third step, the three documents generated in the previous step are considered for the construction of the RSUS data structure. Finally, the mapping of RSUS → JSON Schema is accomplished in the last step. The JSON Schema produced as output is organized in two sections, as explained before: the schema *definitions* section (Figure 5) and the schema *properties* section (Figure 6). The definitions of the first section are referenced in the second section. The second section is the structure of final schema.

An accuracy of 100% was obtained since all expected data types in the input documents were identified. These data types include mandatory attributes, extended JSON types, number of minimum items in an array, and union types when a property has more than one data type (*anyOf*). It is worth highlighting the meaning of the *"additionalProperties"* and *"additionalItems"* attributes in the properties section (Figure 6). The first one is set to *false* because the process is applied to all documents in the collection, i.e., all attributes in the dataset documents are already defined in the schema. The

second one was set to *true* meaning that an array may have more than one item.

B. Processing Time Evaluation

In order to evaluate the processing time of *JSON Schema Discovery*, experiments were conducted on an Amazon EC2 t2.micro instance (Intel® Xeon® E5-2676 v3 @ 2.40GHz and 1GB of RAM). The datasets used in the experiment are tracked data from *tweets*, *checkins* and *venues* from *Foursquare* [14].

Figure 5. JSON Schema - Definitions

```
{
  "$schema": "http://json-schema.org/draft-06/schema#",
  "definitions": {
    "ObjectID": {
      "type": "object",
      "properties": { "$oid": { "type": "string" } },
      "required": [ "$oid" ] },
    "Binary": {
      "type": "object",
      "properties": { "$binary": { "type": "string" },
            "$type": { "type": "string"} },
      "required": [ "$binary", "$type" ]  },
    "Date": {
      "type": "object",
      "properties": { "$date": { "type": "string" } },
      "required": [ "$date" ] },
    ...,
    "DBRef": {
      "type": "object",
      "properties": { "$id": { "type": "string" },
            "$ref": { "type": "string" } },
      "required": [ "$id", "$ref" ] },
    "Long": {
      "type": "object",
      "properties": { "$numberLong": { "type": "string" } },
      "required": [ "$numberLong" ] },
    ...,
    "MaxKey": {
      "type": "object",
      "properties": { "$maxKey": { "type": "integer" } },
      "required": [ "$maxKey" ] } },
  "properties": {...},
  "additionalProperties": false,
  "required": [...]
}
```

Table III
RESULTS FOR FOURSQUARE DATASETS

Collection	N_JSON	RS	ROrd	TB	TT	TB/TT
venues	2 million	257	117	7,47 min	7,52 min	99,33%
checkins	11 million	2	2	35,27 min	35,52 min	99,29%
tweets	17 million	23	16	53,11 min	53,44 min	99,38%

N_JSON - Number of JSON documents. RS - Raw schemas.
ROrd - Raw schemas with ordered structure.
TB - Time to obtain the raw schemas. TT - Total time.

Experimental results are shown in Table III, presenting the average processing time spent in the schema extraction process for the datasets. It is worth mentioning the importance of the ordering of the raw schema attributes, as observed for the *venues* dataset, in which the actual amount of distinct raw schemata obtained was 117, against 257 obtained before the ordering of the attributes.

Another point to observe is that the processing time spent on reading the documents and generating the raw schemas (TB) reached around 99% of the total processing time (TT), since all the documents in a collection must be read. It shows that the bottleneck of our process is dominated by the document reading and not the processing steps themselves.

Figure 6. JSON Schema - Properties

```
{
  "$schema": "http://json-schema.org/draft-06/schema#",
  "definitions": {...},
  "properties": {
    "_id": { "$ref": "#/definitions/ObjectID" },
    "arrayMultiValues": {
      "name": "arrayMultiValues",
      "type": "array",
      "items": { "anyOf": [
          { "type": "number" },
          { "type": "string" },
          { "type": "object",
            "properties": {
              "property": { "name": "property",
                "type": "string" }  },
            "additionalProperties": false,
            "required": [ "property" ] },
          { "name": "arrayMultiValues",
            "type": "array",
            "items": { "anyOf": [
                { "type": "number" },
                { "type": "string" } ] },
            "minItems": 1,
            "additionalItems": true },
          { "type": "boolean" },
          { "$ref": "#/definitions/Binary" } ] },
      "minItems": 1,
      "additionalItems": true },
    "arrayOfArray": {...},
    "arrayOfBoolean": {...}
    ...   },
  ...
}
```

Even so, the increase in the number of considered JSON documents does not increase the time processing spent by our approach to analyse them proportionally.

C. Comparison with Related Work

We also evaluate our approach considering the same *DBPedia* datasets used by the work of Wang et al. [11], which is a recent and very referenced baseline. The experiments were run on an ASUS K45VM notebook (Intel® Core ™ i7 3610QM @ 2.30 GHz and 8GB of RAM). The datasets (see Table IV) were obtained and stored in *MongoDB*. The purpose here is to evaluate the quality of the generated schemas of our approach against the related works. A comparison of processing time against related works will be the focus of future work.

These datasets have many different raw schemas (see *Raw schemas* and *Raw schemas with ordered structure* columns) as the number of schemas is very close to the total of documents in the dataset (see column *Number of documents*). According to Wang et al. [11], this is due to the nature of the dataset (*DBPedia* documents), whose documents usually have some heterogeneities.

Regarding the number of distinct identified schemas, both approaches discovered the same value for the dataset *drugs*, whereas our approach identified a larger number of schemas for *companies* and *movies* datasets. This difference had occurred because *JSON Schema Discovery* takes into account the name and type of each document attribute to define the raw schema, and not only the attribute names, as accomplished by the work of Wang et al. [11]. In fact, the data type is also important to allow more accurate queries with filters on certain attributes.

Table IV
COMPARISON WITH WANG ET AL. [11]

Datasets		**JSON Schema Discovery**		**Wang et al.**[11]
Collection	**N_JSON**	**RS**	**ROrd**	**FS**
drugs	3662	2818	2818	2818
companies	24367	21312	21312	21302
movies	30330	25140	25140	25137

N_JSON - Number of JSON documents. RS - Raw schemas. ROrd - Raw schemas with ordered structure. FS - Final Schemas.

We also compare our approach with the work of Baazizi et al. [13], the most recent related work. We obtained the same number of schemas for the same dataset considered by them. It points out that the accuracy of our approach is promising, being equivalent or superior than related work.

VI. CONCLUSION

This paper introduces an approach called *JSON Schema Discovery*. Its purpose is to extract schemas from JSON or Extended JSON document collections. The main motivation of our work is to contribute with complex tasks such as data integration and retrieval from schemaless data sources, in particular, JSON collections that are usually maintained by NoSQL document-oriented DB. Once a schema from these data sources is discovered, it is possible to know which attributes, data types and constraints are associated to the raw data and, as a consequence, the tasks mentioned before may be accomplished with better accuracy.

JSON Schema Discovery contributes to the state-of-art of schema discovery for JSON data by mainly allowing, at the same time, the automatic schema extraction of both JSON and Extended JSON documents in *JSON schema* format, and the extraction of attribute name, data type and participation constraint (required/optional attribute). Besides, our extraction process considers data summarization operations based on a data structure that maintain several metadata information about the document data for schema extraction purposes, as well as aggregation operations that improves the performance of the schema extraction task. Nevertheless, our approach is available as a free software tool that is able to generate, store and visualize JSON schemas from data collections stored in *MongoDB* NoSQL database.

As future work, we intend to consider: *(i)* Well-formed validation of JSON documents before considering them as input for our schema extraction process; *(ii)* Performance improvement of the first extraction step by adopting parallel processing techniques; *(iii)* Identification and treatment of data relationships, similar to foreign keys of the relational data model. The *JSON Schema Discovery* source code is available in the project page http://lisa.inf.ufsc.br/wiki/index.php/JSON_Schema_Discovery.

REFERENCES

[1] J. Han, E. Haihong, G. Le, and J. Du, "Survey on NoSQL database," in *6th Int. Conf. on Pervasive Computing and Applications*, ser. ICPCA, 2011, pp. 363–366.

[2] R. Cattell, "Scalable SQL and NoSQL data stores," *ACM SIGMOD Record*, vol. 39, no. 4, p. 12, 2011. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1978915.1978919

[3] P. J. Sadalage and M. Fowler, *NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence*, 2012.

[4] H. Hashem and D. Ranc, "Evaluating NoSQL document oriented data model," in *4th Int. Conf. on Future IoT and Cloud Workshops*, ser. W-FiCloud, 2016, pp. 51–56. [Online]. Available: http://ieeexplore.ieee.org/document/7592700/

[5] M. Wischenbart and et al., "User profile integration made easy: Model-driven extraction and transformation of social network schemas," *21st WWW*, pp. 939–948, 2012.

[6] D. S. Ruiz and et al., "Inferring Versioned Schemas from NoSQL Databases and its Applications," *LNCS)*, vol. 9381, no. October, pp. 467–480, 2015. [Online]. Available: http://link.springer.com/10.1007/978-3-319-25264-3_35

[7] F. Pezoa and et al., "Foundations of JSON Schema," in *25th Int. Conf. on World Wide Web*, ser. WWW. WWW Steering Committee, 2016, pp. 263–273. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2872427.2883029

[8] D. C. Schmidt, "Guest editor's introduction: Model-driven engineering," *IEEE Computer*, vol. 39, no. 2, pp. 25–31, 2006. [Online]. Available: https://doi.org/10.1109/MC.2006.58

[9] ECMA-404, "The JavaScript Object Notation (JSON) Data Interchange Format," *ECMA International*, no. Oct., p. 8, 2014. [Online]. Available: https://www.rfc-editor.org/info/rfc7159

[10] M. Klettke and et al., "Schema Extraction and Structural Outlier Detection for JSON-based NoSQL Data Stores," in *BTW*, ser. LNI, vol. 241. GI, 2015, pp. 425–444.

[11] L. Wang and et al., "Schema management for document stores," *Proc. of the VLDB Endowment*, vol. 8, no. 9, pp. 922–933, 2015.

[12] J. L. C. Izquierdo and et al., "Discovering implicit schemas in json data," in *13th Int. Conf. on Web Engineering*, ser. ICWE. Berlin, Heidelberg: Springer-Verlag, 2013, pp. 68–83.

[13] M.-a. Baazizi and et al., "Schema Inference for Massive JSON Datasets," *EDBT*, pp. 222–233, jan 2017.

[14] E. Çelikten, G. L. Falher, and M. Mathioudakis, "Modeling urban behavior by mining geotagged social data," *IEEE Transactions on Big Data*, vol. 3, no. 2, pp. 220–233, June 2017.