An Approach for Schema Extraction of NoSQL Graph Databases

Conference Paper · August 2020

DOI: 10.1109/IRI49571.2020.00046

3 authors, including:

Angelo Augusto Frozza
Instituto Federal de Educação, Ciência e Tecnologia Catarinense (IFC)
27 PUBLICATIONS **143** CITATIONS
SEE PROFILE

Ronaldo Mello
Federal University of Santa Catarina
94 PUBLICATIONS **734** CITATIONS
SEE PROFILE

An Approach for Schema Extraction of NoSQL Graph Databases

Angelo Augusto Frozza
Instituto Federal Catarinense (IFC), Santa Catarina, Brazil
E-mail: angelo.frozza@ifc.edu.br

Salomão Rodrigues Jacinto, Ronaldo dos Santos Mello
Universidade Federal de Santa Catarina (UFSC)
Programa de Pós-Graduação em Ciência da Computação (PPGCC)
Santa Catarina, Brazil
E-mail: maorodriguesj@gmail.com, r.mello@ufsc.br

Abstract

Currently, a large volume of heterogeneous data is generated and consumed by several classes of applications, which raise a new family of database models called NoSQL. NoSQL graph databases is a member of this family. They provide high scalability and are schemaless, i.e., they do not require an implicit schema such as relational databases. However, the knowledge of how data is structured may be of great importance for data integration or data analysis processes. There are some works in the literature that extract the schema from graph structures or graph-based data sources. Different from them, this work proposes a comprehensive approach that consider all the common NoSQL database graph data model concepts, and generates a schema in the recent JSON Schema recommendation. Experimental evaluations show that our solution generates a suitable schema representation with a linear complexity.

1. Introduction

NoSQL databases is a relatively new family of database management system (DBMS) suitable to *Big Data* management as they present flexible data models as well as high availability and scalability [8, 12]. *Schemaless* is another usual feature of them. Most of the NoSQL products do not obligate a schema to be defined *a priori* to a database, as relational databases do. They argue that this feature avoids a rigid data structure control, which would impose a management overhead to the DBMS.

Nevertheless, to be schema-aware is important to several data management tasks, like data integration, data interoperability, as well as query processing and optimization. One example is the recent data lake management system [9]. A data lake is a huge repository of heterogeneous data sources that must frequently be integrated in order to answer a data analytics demand. This task is hard to be accomplished without a metadata management, which requires a schema extraction or a reverse engineering process [4].

Some efforts in the Database research community try to propose solutions to the problem of schema extraction of NoSQL databases [5, 7, 11, 15]. However, this problem is still an open issue for most of the NoSQL data models, as is the case of NoSQL graph databases [10]. There are some few proposals in the literature, but they have several limitations as they do not consider all the concepts of the NoSQL graph data model and/or propose complex schema extraction processes without any evaluation [2, 3, 13, 14].

This paper proposes a more robust approach for schema extraction of NoSQL graph databases. It introduces an extraction process that takes into consideration all the concepts available at the data models of the most popular NoSQL graph DBMS, and a factoring strategy that generates a short schema as the result of the process. Other contributions of our solution are an experimental evaluation regarding quality of the generated schema and processing time performance, as well as an output schema described in *JSON Schema*[1]. *JSON Schema* is a recent and well-accepted recommendation for data representation and exchanging. Our approach is detailed in the next sections.

The rest of the paper is organized as follows. Section 2 introduces NoSQL graph database and formalize its data model. Section 3 presents our approach, including the proposed steps of our schema extraction process. Section 4 describes our experimental evaluation. Section 5 discusses the related work and Section 6 is dedicated to the conclusion.

[1]https://json-schema.org

978-1-7281-1054-7/20/$31.00 ©2020 IEEE
DOI 10.1109/IRI49571.2020.00046

2 NoSQL Graph Databases

A NoSQL graph database is usually a *property graph* of data, *i.e.*, a graph structure where vertices may hold properties [1, 10, 12]. Despite this consensus, there is not a standard data model adopted by the most popular native NoSQL graph DBMS[2], as shown in Table 1.

Vertex and edge are basic concepts of a graph structure, as well as vertex properties for a property graph. A vertex usually models a real-world object, an edge represent a relationship between two vertexes, and a property is an attribute-value pair where the value is restricted to a datatype. Most of the data models also provide additional details for edges: properties, label and relationship direction. Labels for vertexes usually denote a semantic type for the object. Additionally, some data models support multiple labels for vertexes, in sense that an object belongs to more than a type (*e.g., actor* and *director*). Finally, it is possible that some properties hold complex datatypes (*e.g., lists, tuples, geographic types*) besides traditional atomic datatypes (*e.g., integer, string*).

We argue that a comprehensive NoSQL graph database supports all of these data model concepts shown in Table 1. Based on this assumption, we formalize a NoSQL graph database, as well as its components, in the following.

Definition 1 ***(NoSQL Graph Database)****. A NoSQL graph database is a tuple gdb* $= (n_{db}, V, E)$*, where* n_{db} *is the name of the database,* V *is the set of vertexes,* $|V| >= 0$*,* E *is the set of edges,* $|E| >= 0$*, and gdb is identified by* n_{db}*.*

Definition 2 ***(Vertex)****. A vertex* $v_i \in gdb.V$ *is a tuple* $v_i = (P, L)$*, where* P *is the set of vertex properties,* $|P| >= 0$*, and* L *is the set of vertex labels,* $|L| >= 0$*.*

Definition 3 ***(Edge)****. An edge* $e_k \in gdb.E$ *is a tuple* $e_k = (source, target, label, directed, P)$*, where* $source \in gdb.V$ *is the edge source vertex,* $target \in gdb.V$ *is the edge target vertex,* $label$ *is the edge label,* $directed$ *is a boolean property denoting whether* e_k *is directed or not, and* P *is the set of edge properties,* $|P| >= 0$*.*

Definition 4 ***(Property)****. A property* $p_j \in V.P \cup E.P$ *is a tuple* $p_j = (n_p, v_p)$*, where* n_p *is the property name, and* v_p *is the property value with an atomic or complex datatype.*

If the graph database is not directed then there are not source and target roles for the vertexes. In this case, the two related vertexes may be associated with any of these two edge properties (*source* and *target*).

Our approach considers the schema extraction of a NoSQL graph database as stated in Definition 1. It is detailed in the next section.

[2]https://db-engines.com/en/ranking/graph+dbms

3 Proposed Approach

Our approach is proposed as a process that receives as input a NoSQL graph database *gdb* and returns as output an inferred schema for *gdb*. We chose *JSON Schema* as the output format because JSON[3] is a current and well-adopted standard for complex data representation and data interchange, and JSON Schema is a *de facto* recommendation for representing the schema of JSON data.

Figure 1 shows our schema extraction process, which comprises three steps. The first step (*Grouping*) scans the *gdb.V* and generates a group for each different *powerset* element $ps_i \in PS$ of labels in $gdb.V.L$. A group (a set of vertexes with the same labels) maintains information about properties and edges of the vertexes that belongs to the group. This step also scans the *gdb.E* and generates groups of edges with the same label. A group maintains information about the properties of its edges. The second step (*Factoring*) scans and analyzes PS in order to generate sets of individual labels with their specific properties and edges. Finally, the *Parsing* step generates a set of *JSON Schema* documents that summarizes all the schema information about vertexes and edges.

A novelty of our approach is the treatment of the multiple labels of a vertex, as permitted by Definition 2. This is the task of the *Factoring* step. By using a strategy based on set operations, it identifies the specific label on which a property belongs when a vertex has more than one label. It provides a property factoring and the generation of an inheritance hierarchy that reduces the size of the inferred schema. We detail all of these steps in the following.

3.1 Grouping Step

The *Grouping* step receives as input a NoSQL graph database and outputs a set of groups of vertexes and edges with the same label set or label, respectively. Algorithm 1 describes the tasks performed by this step for generating groups of vertexes.

The core of Algorithm 1 is the *Powerset* function (line 3). It receives a list of vertex labels and generates a set with all possible subsets of labels. Suppose a NoSQL database graph *gdb = ('Movies',* $\{v_1, v_2\}$, $\{e_1\}$*)*, where v_1 = *(*$\{$*(name,'Chris Evans'), (oscar, false)*$\}$, $\{$*'Person','Actor'*$\}$*)*, v_2 = *(*$\{$*(name,'Avengers'), (genre,'adventure')*$\}$, $\{$*'Movie'*$\}$*)* and e_1 = *(*v_1,v_2*,'acts_in', true,* $\{\}$*)* in a *Movies* domain. As *gbd* has three vertex labels, we have: ***Powerset****([Person,Actor,Movie])* = $\{\{\}$, $\{$*Person*$\}$,$\{$*Actor*$\}$,$\{$*Movie*$\}$,$\{$*Person, Actor*$\}$,$\{$*Person, Movie*$\}$,$\{$*Actor, Movie*$\}$,$\{$*Person, Actor, Movie*$\}\}$.

Algorithm 1 first scans all graph vertexes to find out the database list of labels L_l, and then apply the *Powerset* func-

[3]https://www.json.org

Table 1. Data model concepts adopted by NoSQL graph DBMS.

Data Model Concept	GraphDB	Neptune	DGraph	JanusGraph	Neo4j
vertex	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$
edge	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$
vertex property	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$
edge property		$\surd$	$\surd$	$\surd$	$\surd$
directed edge	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$
vertex label	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$
edge label	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$
multiple label for vertex		$\surd$			$\surd$
complex property	$\surd$		$\surd$	$\surd$	$\surd$

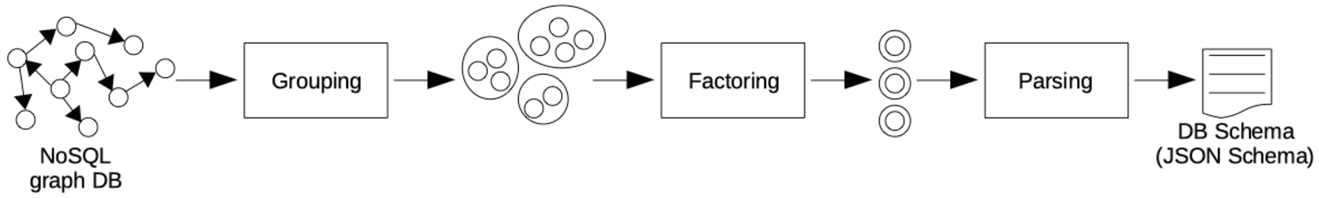

Figure 1. Schema extraction process

tion over L_l (lines 2 and 3). Then, for each subset $SubS_l$ of labels (line 5) it retrieves the list of vertexes L_v that belong to $SubS_l$ (line 6), and also stores the list size ($|L_v|$) (line 7). Line 9 defines the grouping for L_v, and the inner loop (line 9 to 12) extracts the properties and edges of the vertexes in L_v. The last task is to check if a property p_i must be set as mandatory or not. It is true if the number of p_i occurrences is equal to $|L_v|$ (line 13).

For the gbd NoSQL database, one example of group generated by this step is *(Person, Actor)* with properties: $\{$*('name','string',true), ('oscar','boolean',false)*$\}$ and edges: $\{$*('acts_in','Movie',true)*$\}$. It shows that vertexes with these two labels hold a mandatory property $name$, a mandatory edge $acts\_in$ that points to a $Movie$ vertex, and an optional property $oscar$. This step also infer the datatype of each property based on the more generic datatype that matches all property values.

A process similar to Algorithm 1 is also performed to group edges per label and infer mandatory and optional properties for them. For sake of paper space, we do not detail it.

3.2 Factoring Step

The factoring step analyzes the groups of vertexes in order to convert groups of vertexes with more than one label into groups of vertexes with only one label. The purpose here is to generate an inheritance hierarchy (a type hierarchy) of vertexes types so that generic properties and edges are defined once and are inherited to all specialized vertexes types that hold them. Besides reducing the amount of properties and edges to be specified in the inferred schema, it provides a more semantic meaning for the schema.

For sake of understanding, suppose the following set of vertexes groups generated by the *Grouping* step for a NoSQL database graph in a $Movies$ domain:

- *(Actor, Person): properties:*$\{$*('name', 'string', true), ('oscar', 'boolean', false)*$\}$*; edges:*$\{$*('acts_in', 'Movie', true)*$\}$;
- *(User, Person): properties:*$\{$*('name', 'string', true), ('lastLogin', 'string', true)*$\}$*; edges:*$\{$*('rated', 'Movie', false)*$\}$;
- *(Movie): properties:*$\{$*('name', 'string', true), ('genre', 'string', true)*$\}$.

Algorithm 2 rules this step. It is based on a loop (line 4) that generates single label vertexes groups while it consumes the input. A boolean variable (line 3) controls the end of the factoring process.

Single label vertexes groups are initially moved to a temporary list (lines 8 to 13) and forward to the output list ($unique\_labels$ - lines 15 to 19) as they are factorized. In our example, the $Movie$ group is inserted into $unique\_labels$.

Lines 21 to 32 controls the factoring execution. It initially applies a set intersection operation over the input groups to find out labels that belong to more than one group (line 21). In our example, we generate *Person: [*$\{$*Actor,*

Algorithm 1: Vertexes Grouping

```
   Input: NoSQL graph DB
   Output: groups of vertexes per label set
1  begin
2      $labels \longleftarrow$ list of vertexes labels;
3      $labels\_powerset \longleftarrow Powerset(labels)$;
4      $grouping \longleftarrow \{\}$;
5      for $label\_combination \in labels\_powerset$ do
6          $records \leftarrow$ list of vertexes with
             $label\_combination$;
7          $key \leftarrow (label\_combination,$ records size$)$;
8          $grouping[key] \longleftarrow \{\}$;
9          for $rec \in records$ do
10             $Process\_properties(grouping, key, rec)$;
                 $Process\_edges(grouping, key, rec)$
11         end
12     end
13     $Check\_mandatory\_properties(grouping)$;
14     return grouping
15 end
```

Algorithm 2: Factoring of Groups of Vertexes

```
   Input: groups of vertexes (group_nodes)
   Output: groups of vertexes with a single label
1  begin
2      $unique\_labels \longleftarrow \{\}$;
3      $labels\_to\_process \longleftarrow True$;
4      while $labels\_to\_process$ do
5          $labels\_to\_process \longleftarrow False$;
6          $labels\_mix \leftarrow group\_nodes.keys$;
7          $labels\_length\_1 \leftarrow \{\}$;
8          for $label \in labels\_mix$ do
9              if $label.length == 1$ then
10                 $unique\_labels[label] \longleftarrow$
                     $group\_nodes.pop(label)$;
11                 $labels\_to\_process \leftarrow True$;
12                 $labels\_length\_1.add(label)$;
13             end
14         end
15         if $labels\_to\_process == True$ then
16             for $label \in labels\_length\_1$ do
17                 $group\_nodes \leftarrow$
                     $Process\_intersect(group\_nodes,$
18                 $label, unique\_labels[label])$;
19             end
20         else
21             $intersections \longleftarrow$
                 $Labels\_intersect(labels\_mix)$;
22             if $intersections.length > 0$ then
23                 $labels\_to\_process \leftarrow True$;
24                 $key\_intersect \leftarrow$node type with
                     most intersections;
25                 $unique\_labels[key\_intersect] \longleftarrow$
                     $Intersect\_properties(group\_nodes,$
26                 $key\_intersect,$
27                 $intersections[key\_intersect])$;
                     $group\_nodes \leftarrow$
28                 $Process\_intersect(group\_nodes,$
29                 $key\_intersect,$
30                 $unique\_labels[key\_intersect])$;
31             end
32         end
33     end
34     if $group\_nodes.length > 0$ then
35         for $key \in group\_nodes$ do
36             $unique\_labels[key] \longleftarrow$
                 $group\_nodes[key]$;
37         end
38     end
39     return unique_labels
40 end
```

Person}, {User, Person}] as this label occurs at these two groups. In case of intersections exist (line 22), the factoring process selects the label l_{max} with the biggest number of intersections (line 24) and a group is generated for l_{max} with the common properties and edges in all groups l_{max} participates (lines 25 to 27). In our example, we have *Person: properties: {('name', 'string', true)}*.

In the following, the function $Process\_intersect$ (lines 28 to 30) reorganizes the remaining vertexes groups to consider the previous factoring. It removes l_{max} from the set of vertexes groups VG_i it appears, as well as its common properties and edges. Besides, it specifies, for all $vg \in VG_i$, that l_{max} is a supertype of them. In our example, it removes the groups where the label *Person* exist, and generates the following groups: *Actor: properties: {('oscar', 'boolean', false)}, edges: {('acts_in', 'Movie', true)}, supertype: {Person}; User: properties: {('lastLogin', 'string', true)}, edges: {('rated', 'Movie', false)}, supertype: {Person}*. The loop of line 4 goes on until all vertexes groups of multiple labels be processed and the type hierarchy be completely defined. At the end (lines 34 to 38), the remaining vertexes groups with single labels are added to the $unique\_labels$ list. Besides the generated *Person, Actor* and *User* groups, this step also outputs the *Movie* group, as stated before.

3.3 Parsing Step

The parsing step is responsible to produce a JSON Schema document for each single inferred vertex and edge schema. We decided to generate separated JSON Schema documents instead of a large single one for sake of understanding of the graph schema as well as to facilitate the reuse of parts of the whole NoSQL graph database schema. The parsing process also takes into consideration the NoSQL graph database to JSON Schema datatype compatibility table (Table 2) to generate appropriate datatypes in the output schema.

Table 2. NoSQL DB-JSON Schema datatype compatibility

NoSQL graph DB	JSON Schema
null	null
integer	integer
float	number
string	string
list	array
map	object
boolean	boolean

Examples of JSON Schema documents for our *Movies* domain example are shown in Section 4.1 for the IMDB database (Figures 6 and 7). For each vertex and edge document an id property is included in order to identify them, and the JSON Schema `allOf` element is used to describe *supertype-subtype* relationships.

4 Experimental Evaluation

This section presents two evaluations of the proposed approach. The first one analyzes the quality of the schema extraction over two real NoSQL graph databases, and the second one evaluates the processing time spent by the approach to execute several schema extractions.

In order to concretely evaluate our approach, we developed a prototype system that extracts schemas from the Neo4j DBMS. We consider Neo4j as it has the more robust NoSQL graph data model, as shown in Table 1.

4.1 Qualitative Evaluation

Two public NoSQL graph databases were considered for this evaluation: *Airbnb*[4] and *IMDB*[5]. The first one is a service for managing lodgings, and the second one is a database about movies.

For *Airbnb*, we first execute the Neo4j *db.schema()* function, which generates a graph with vertex and edge types in the database, as shown in Figure 2. This is an incomplete schema, as no properties are presented.

[4]https://www.airbnb.com
[5]https://www.imdb.com/

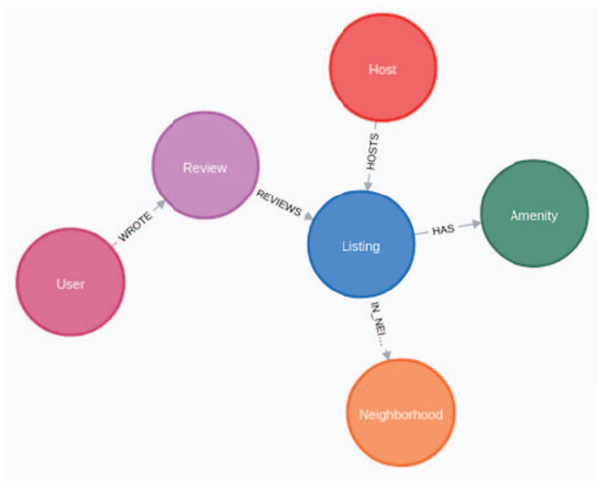

Figure 2. Airbnb Schema

As Neo4j does not provide detailed schema information, we had to accomplish a manual analysis of the database structure. For sake of paper space, we focus here on the *Host* vertex and the *Hosts* edge, but the analysis could be extended to other parts of the database. From the analysis, we discovered that any *Host* holds the *name, superhost, image, location* properties, and it has at least one *Hosts* edge.

Figures 3 and 4 present the generated schema for the *Hosts* edge and *Host* vertex by our approach, respectively. It correctly retrieves all the expected *Host* properties, and all of them are defined as required. Figure 4 also shows that the *Host* vertex must hold a *Hosts* edge to the *Listing* vertex (see Figure 2), as its schema has 1 as value for the *"minItems"* constraint.

These results demonstrate that our approach had obtained a 100% accuracy as the *Host* and *Hosts* schemas were completely inferred. In fact, we got this excellent accuracy for all the database schema.

```
{   "$schema": "https://json-schema.org/
    draft/2019-09/schema#",
    "title": "HOSTS",
    "description": "HOSTS relationship",
    "$id": "hosts.json",
    "type": "object",
    "properties": {
      "<id>": {"type": "number"}},
    "required": ["<id>"],
    "additionalProperties": false}
```

Figure 3. JSON Schema for *Hosts* edge

For *IMDB*, we can see the result of Neo4j *db.schema()* function in Figure 5. Based also on a manual analysis of the database instances, we see that this structural summary

```
{   "$schema": "https://json-schema.org/
    draft/2019-09/schema#",
    "title": "Host",
    "description": "Host node",
    "$id": "host.json",
    "definitions": {
      "hosts": {"type": "object",
                "$ref": "hosts.json"},
      "listing": {"type": "object",
                  "$ref": "listing.json"}},
    "type": "object",
    "properties": {
        "<id>": {"type": "number"},
        "name": {"type": "string"},
        "superhost": {"type": "boolean"},
        "image": {"type": "string"},
        "location": {"type": "string"},
        "relationships": {
          "type": "array",
          "items": {
          "oneOf": [
          {"type": "array",
           "items": [
            {"type":"object",
             "$ref":"#/definitions/hosts"},
            {"type":"object",
             "$ref":"#/definitions/listing"}
          ]}]},
          "minItems": 1}},
    "required": ["<id>","name","superhost",
    "image","location","relationships"],
    "additionalProperties": false}
```

Figure 4. JSON Schema for *Host* vertex

could be better described. For example, there are not vertexes that hold only *Person* or *Actor* labels. In fact, all instance vertexes belong to one of these label combinations: *Movie*, {*User, Person*}, {*Actor, Person*}, {*Director, Person*} and {*Actor, Director, Person*}.

Thus, our factoring strategy for vertex labels is suitable to this database. It generates a type hierarchy so that the vertex schema with the *Person* label holds properties and edges that are common to the vertex schemas with the *Actor*, *User* and *Director* labels. This strategy reduces the size of the extracted schema since it factorizes and represents once the common properties, as can be shown in the output schemas generated by our approach (Figures 6 and 7)[6]. Besides this optimization, our approach also obtained a 100% accuracy in terms of schema inference.

[6]*Actor* schema definition holds a constraint denoting that all actor instances are also person instances (`allOf`).

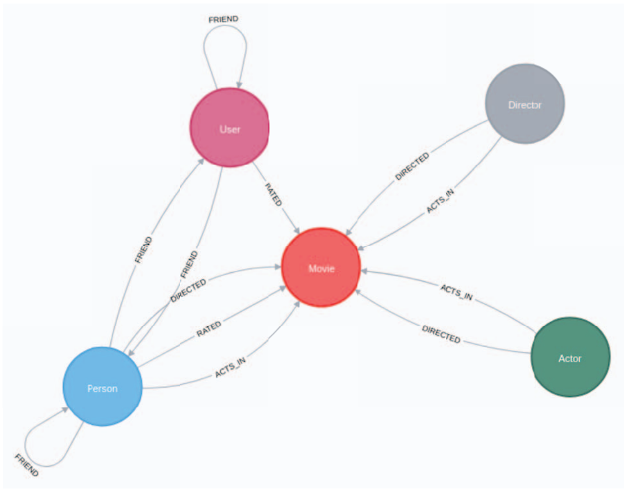

Figure 5. IMDB schema

```
{"$schema": "https://json-schema.org/
 draft/2019-09/schema#",
    "title": "Person",
    "description": "Person node",
    "$id": "person.json",
    "definitions": {},
    "type": "object",
    "properties": {
        "<id>": {"type": "number"},
        "name": {"type": "string"}},
    "required": ["<id>","name"],
    "additionalProperties": false}
```

Figure 6. JSON Schema for *Person* vertex

4.2 Performance Evaluation

This section presents a time execution evaluation for schema extraction of Neo4j databases accomplished by our prototype. For this experiment, we selected four graph databases with different number of vertexes and edges. Besides *Airbnb* and *IMDB* (see Section 4.1), we also consider a TV series database (*Dr Who*[7]) and a large database extracted from the *Stackoverflow* API[8]. *Stackoverflow* is a repository about questions and answers related to programming languages. The number of vertexes and edges for each database are presented in Table 3.

We ran the experiments over an Intel(R) Core(TM) i5-7400 CPU @ 3.00GHz 16Gb RAM computer with Arch Linux operating system and a non-volatile SSD. Each database was hosted inside a docker[9] container of Neo4j version 3.5.12, and our prototype was executed with Python

[7]https://gist.github.com/fbiville/b875f5bf1e4052f7c22927ffd5eda1f9
[8]https://stackoverflow.com/
[9]https://www.docker.com/

```
{"$schema": "https://json-schema.org/
    draft/2019-09/schema#",
    "title": "Actor",
    "description": "Actor node",
    "$id": "actor.json",
    "definitions": {
      "acts_in": {"type": "object",
                  "$ref": "acts_in.json"},
      "movie": {"type": "object",
                "$ref": "movie.json"}},
    "type": "object",
    "allOf": [{"$ref": "person.json"}],
    "properties": {
     "id": {"type": "string"},
     "birthplace": {"type": "string"},
     "version": {"type": "number"},
     "profileImageUrl": {"type": "string"},
     "biography": {"type": "string"},
     "lastModified": {"type": "string"},
     "birthday": {"type": "string"},
     "relationships": {
       "type": "array",
       "items": {
       "oneOf": [
       {"type": "array",
        "items": [
         {"type": "object",
          "$ref": "#/definitions/acts_in"},
         {"type": "object",
          "$ref": "#/definitions/movie"}]}]
       }, "minItems": 1}},
    "required": ["id","birthplace",
    "version",
    "profileImageUrl","biography",
    "lastModified","birthday",
    "relationships"],
    "additionalProperties": false}
```

Figure 7. JSON Schema for *Actor* vertex

version 3.7.4.

The experiment results show that the processing time spent by our prototype increases linearly w.r.t. the number of vertexes and edges since the groupings provided by our *powerset* function avoid that a vertex be visited more than once for each identified label. So, the computational complexity of our approach is *O(#V+#E)*. If each vertex had to be visited in order to find out each label schema, the worst case complexity would be *O(#L.#V+#E)*, where *#L* is the number of labels in the graph database. This is clearly a pretty much bad complexity.

We can demonstrate the computational complexity of our approach by the average processing time taken from ten executions of our prototype over each one of the four databases. Table 4 shows the averages per database, which were plotted into a line graph (Figure 8) that highlights the linear behavior of our approach w.r.t. the number of graph vertexes and edges.

Table 3. Graph database sizes

Database	#V	#E
Dr Who	1.060	2.286
IMDB	63.042	106.651
Airbnb	129.444	220.183
Stackoverflow	690.656	1.408.205

Table 4. Average processing time for schema extraction

Database	Average processing time
Dr Who	1.064 seconds
IMDB	53.084 seconds
Airbnb	108.935 seconds
Stackoverflow	556.656 seconds

5 Related Work

There are approaches in the literature that are related to this work. Some of them introduce schema extraction processes for general graph structures, having no focus on NoSQL graph databases, *i.e.*, they do not consider graphs whose vertices and edges have properties [13, 14]. Another one deals with the problem of schema inference for ontologies [6]. Its purpose is also different as it identifies and extracts axioms and constraints.

Two other works have similar purposes as they are directed to the schema extraction of Neo4J databases. Nevertheless, they have several limitations. The first one do not present an extraction process as well as an output format to be persisted and queried [2]. Instead, it suggests only a set of Cypher queries to extract vertex and edge labels in order to be aware of their data types. The second one proposes a reverse engineering process that generates a conceptual schema from a property graph [3]. The process, however, is very complex (more than 70 conversion rules), and it is neither detailed in terms of algorithms nor evaluated through experiments to verify quality and performance issues.

6 Conclusion

This paper introduces an approach for schema extraction of NoSQL graph databases. Schema extraction of NoSQL DBMS is an important task as this kind of database is gaining momentum and the need for integration and interoper-

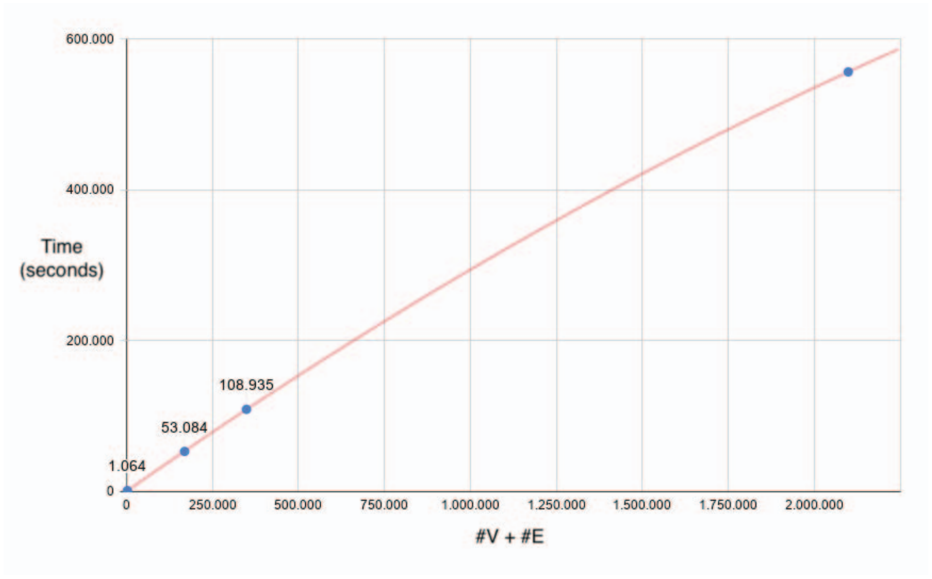

Figure 8. Processing time vs. Graph database size

ability of NoSQL data sources is becoming relevant. Compared to related work, our approach innovates by considering a comprehensive NoSQL graph data model on which the schema extraction process works, a strategy that factorizes common parts of the inferred schema in order to reduce its size, an output schema represented in *JSON Schema*, which raises as a standard for NoSQL data interoperability, as well as an experimental evaluation that demonstrates its viability.

Future works include experimental evaluations considering larger and different datasets as well as other NoSQL graph DBMS, a qualitative and performance comparison against the related work, and a deep analysis of our algorithms in order to leverage their performance. We also intend to include these approach into a more complex solution dedicated to the integration of NoSQL databases.

References

[1] R. Angles. A Comparison of Current Graph Database Models. In *International Conference on Data Engineering Workshops*, pages 171–177. IEEE, 2012.

[2] A. Castelltort and A. Laurent. Extracting Fuzzy Summaries from NoSQL Graph Databases. In *11th International Conference FQAS*, volume 400 of *Advances in Intelligent Systems and Computing*, pages 189–200. Springer, 2015.

[3] I. Comyn-Wattiau and J. Akoka. Model Driven Reverse Engineering of NoSQL Property Graph Databases: The Case of Neo4j. In *International Conference on Big Data*, pages 453–458. IEEE, 2017.

[4] X. L. Dong and D. Srivastava. *Big Data Integration*. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2015.

[5] A. A. Frozza, R. S. Mello, and F. S. Costa. An Approach for Schema Extraction of JSON and Extended JSON Document Collections. In *International Conference on Information Reuse and Integration*, pages 356–363. IEEE, 2018.

[6] Q. Ji, G. Qi, H. Gao, and T. Wu. Survey on Schema Induction from Knowledge Graphs. In *3rd China Conference on Knowledge Graph and Semantic Computing*, volume 957, pages 136–142. Springer, 2018.

[7] M. Klettke and et al. Schema Extraction and Structural Outlier Detection for JSON-based NoSQL Data Stores. volume 241 of *LNI*, pages 425–444. GI, 2015.

[8] A. Oussous et al. Comparison and Classification of NoSQL Databases for Big Data. *International Journal of Database Theory and Applications*, 6(4), 2013.

[9] F. Ravat and Y. Zhao. Data Lakes: Trends and Perspectives. In *International Conference on Database and Expert Systems Applications*, pages 304–313, 2019.

[10] I. Robinson, J. Webber, and E. Eifrem. *Graph Databases*. O'Reilly, 2 edition, 2015.

[11] D. S. Ruiz and et al. Inferring Versioned Schemas from NoSQL Databases and its Applications. *LNCS*, 9381:467–480, 2015.

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

[13] Y. Sekine and N. Suzuki. Extracting Schemas from Large Graphs with Utility Function and Parallelization. *LNCS*, 10829:125–140, 2018.

[14] Y. Tsuboi and N. Suzuki. An Algorithm for Extracting Shape Expression Schemas from Graphs. In *Symposium on Document Engineering*, pages 32:1–32:4. ACM, 2019.

[15] M. Wischenbart and et al. User Profile Integration Made Easy: Model-driven Extraction and Transformation of Social Network Schemas. In *21st World Wide Web Conference*, pages 939–948. ACM, 2012.