Schema inference for property graphs

Hanâ Lbath, Angela Bonifati, Russ Harmer

Schema Inference for Property Graphs

1st Hanâ Lbath
Lyon 1 University, CNRS Liris
UdL, CNRS, ENS Lyon, UCBL1
Lyon, France
hana.lbath@ens-lyon.fr

2nd Angela Bonifati
Lyon 1 University, CNRS Liris
Lyon, France
angela.bonifati@univ-lyon1.fr

3rd Russ Harmer
UdL, CNRS, ENS Lyon, UCBL1
Lyon, France
russell.harmer@ens-lyon.fr

Abstract—**Property graph instances are typically populated without defining a schema beforehand. Although this ensures great flexibility, the lack of a schema implies to miss opportunities for query optimization, data integration and analytics, to name a few. Since several graph instances exist prior to the schema definition, extracting the schema from those instances in a principled way might become a significant yet daunting task. In this paper, we present a novel end-to-end schema inference method for property graph schemas that tackles complex and nested property values, multi-labeled nodes and node hierarchies. Our method consists of three main steps, the first of which builds upon Cypher queries to extract the node and edge serialization of a property graph. The second step builds over a MapReduce type inference system, working on the serialized output thereby obtained during the first step. The third step analyzes subtypes and supertypes to infer node hierarchies. We describe our schema inference pipeline and its implementation, a labels- and a properties-oriented variant. Finally, we experimentally evaluate and compare the scalability and accuracy of our approaches on several real-life datasets. To the best of our knowledge, our work is the first to tackle the problem of schema inference for property graphs.**

Index Terms—**Big Graph management, property graphs, schema inference, graph databases, graph subtyping**

I. Introduction

Over the past decade, graph-based knowledge representation has been becoming increasingly popular, be it with the advent of graph databases [1] as an alternative to relational databases, or to model complex systems, from social and fraud detection networks to smart city grids and brain and biological networks.

According to the 2019 Gartner's report on the top 10 Big Data technology trends in the upcoming years, the application of graph processing and graph databases will grow at 100% annually to accelerate data preparation and to enable more complex data science. Property graphs (PG) are one of the main ingredients of this technology, being directed labeled graphs where nodes and edges may store attributes in the form of properties (i.e., key-value pairs). In view of that, property graphs convey greater expressive power compared to edge-labeled graphs. Many modern graph database vendors, such as Neo4j, Tigergraph, Oracle PGX to name a few, have adopted the PG data model as the foundational data model for enabling graph querying and analytical tasks. PGs have also been used in rule-based modeling of complex systems, like cellular signaling networks [2]. However, due to the lack of a standard schema, PG instances are typically built without a predefined schema. Although it ensures great flexibility, it can also become a great impediment, notably whenever the structure of the underlying instance is stabilized. Indeed, schemas succinctly represent the structure of the PG instances and allow to set constraints, such as the types of nodes, edges and properties as well as the cardinality of a relationship or property value data types. Moreover, schemas can be used to build relevant graph features needed in many Machine Learning pipelines [3]. Yet, given that PG instances usually exist prior to the schema definition, extracting a schema from those instances in a principled way might become a significant yet daunting task. To the best of our knowledge, the few PG schema inference methods available can only output basic schemas that cannot handle complex data types, overlapping node types or node hierarchies (cf. Section II). A principled approach to PG schema inference is currently missing and urgently needed given the interest in an ongoing ISO SC32/ WG3 standardization process of PG schemas, involving people from academia and industry in the GQL community [1].

In this work, we present a novel end-to-end schema inference method for PGs where we extend a formal definition of PG schemas to include the notion of inheritance edge types. Our method tackles complex and nested property values, multi-labeled and unlabeled nodes, node hierarchies and overlapping node types. We also infer edge cardinality constraints and optionality of properties. Our pipeline inputs a PG stored in Neo4j and outputs a PG schema stored in a JSON file that can be loaded into Neo4j for visual inspection. In order to enable scalability, our method leverages a MapReduce approach [4] developed for schema inference from JSON datasets, which both aggregates types and identifies data types. The main challenge resides in the conversion of the input data, which is a PG, into a format that will guarantee proper node and edge type inference. Furthermore, subtyping is not addressed in [4] due to the remarkable differences between property graphs and JSON documents and requires special treatment. Moreover, we introduce two variants of our method, i.e. a labels-oriented variant and a properties-oriented one and we investigate their pros and cons.

The schema inference pipeline we designed can be divided

[1]More information can be found at: https://www.gqlstandards.org/

into three steps [2]. First, we employ Cypher queries to extract and serialize the nodes and edges of the input PG, in addition to gathering the information needed to infer edge cardinality constraints. Cypher is an open-source graph query language developed by Neo4j, inspired from SQL and adopted by several graph database vendors. It enables the storage and retrieval of data through graph pattern matching. Afterward, we infer node and edge types, together with the property value data types, using the output from the first step to input the MapReduce algorithm. The last step consists in analyzing subtypes and supertypes to infer node hierarchies. Our schema inference method is generic and can be adapted to other graph database platforms, insofar as the input PG can be appropriately serialized to JSON. Furthermore, our work can serve as a basis to inform the ongoing discussion of the working groups within the ISO SC32/ WG3 standardization process about the impact of the schema inference process on the PG schema design choices.

After describing our schema inference method, we experimentally evaluate the accuracy and scalability of our schema inference method on real-life graph datasets.

Outline. *This paper is organized as follows. We first survey related works in Section II. In Section III, we define the main concepts used in this work alongside the introduction of the inheritance edge types notion in the PG schema definition. In Section IV, we describe our schema inference pipeline and its labels-oriented and properties-oriented versions. Finally, we empirically evaluate our implementation on real-life datasets in Section V, before concluding in Section VI.*

II. Related Work

Several companies propose graph databases supporting the PG data model, such as Neo4j, Oracle PGX, TigerGraph and GraphQL. Neo4j offers the possibility to view a schema of the database via the following Cypher query: `call db.schema`. It outputs a single-labeled directed graph, enabling the user to visualize which types of nodes can be connected together and through which types of edges. However, multi-labeled nodes in the graph instance are duplicated in the graph schema so that each node is assigned a single label, hence loosing label co-occurrence information. Furthermore, properties, edge cardinality constraints and node type hierarchies are not included. Nonetheless, another Cypher query, `call db.schema.nodeTypeProperties` (or `call db.schema.relTypeProperties`), outputs, for each node (or edge) type its corresponding properties and their data types, in addition to whether or not they are mandatory. Nevertheless, in the case of multi-valued or nested properties, only the data type of the data structure containing them is inferred.

In addition, GraphQL schemas can be inferred from Neo4j databases [5], [6] via a Neo4j Desktop GraphQL plugin or neo4j-graphql-js. Node and edge types and node properties data types are inferred, unlike overlapping types, node hierarchies and nested property values—in contrast with our method. Furthermore, some Neo4j-specific data types, such as Locations or Dates, produce an error.

Many schema inference approaches consist in identifying structural graph summaries by grouping *equivalent* nodes together [7]. Typical examples are clustering techniques, such as [8] and [9], which infer *types* in RDF datasets. Both are based on the assumption that the more *properties* two *entities* share, the more likely they belong to the same *type*. To this end, they group entities according to a similarity metric. In addition, they both handle hierarchical and overlapping types. In [9], a density-based clustering method, DBSCAN, is adopted. In [8] a faster and more accurate clustering method, called StaTIX, is proposed. It uses the cosine similarity metric and is based on the Louvain community detection algorithm [10]. However, none of these techniques infer data types.

Up to a certain point, PGs can be encoded in the JSON format. In [4], the authors propose a scalable MapReduce approach for schema inference in JSON datasets, which infers all data types before merging types according to an equivalence relation. Our pipeline repurposes it for PG type inference. However, in JSON, type hierarchies only exist in a very limited capacity and [4] does not tackle explicitly the problem of overlapping types.

None of these approaches fully satisfy our criteria for PG schema inference, i.e. inference of basic types and data types as well as complex data types, overlapping types and node hierarchies, and, to the best of our knowledge, this is the first paper presenting a schema inference method specifically tailored to PGs.

III. Preliminaries

In this section, we define the main concepts used in this paper. We consider a **type** to be a group of objects sharing characteristics. **Labels** can be used to name types. We define a **data type** as the nature of the values data can take, such as `string`. A **property** is a key-value pair held by an object. In this work, we adopt the **Property Graphs** (PG) definition from [1, chap. 2]. In essence, a PG is a directed graph where nodes and edges can be labeled and hold properties, considered either as mandatory or optional.

We also need to define **Property Graph Schemas**, which is in the process of being standardized by the ISO. Several definitions already exist. Angles [11] proposed one based on that of PGs and mentions mandatory properties and cardinality constraints. Nevertheless, it does not handle type hierarchies. GraphQL was also used to define schemas for PGs [5]. However, the schema itself is not a PG and hierarchies are not considered either. TigerGraph introduced a PG schema definition as well [12]. Even though they do not consider cardinality constraints, they handle node type inheritance. However, in their framework, a type can only inherit from a single supertype, which is problematic when dealing with overlapping types. Nonetheless,

[2]Our PG schema inference method is available at https://gitlab.com/Hgit/pgsinference

in relational database schemas, multiple inheritance can be defined [13].

In [14], another PG schema definition is proposed. It is based on discussions in Neo4j and both mandatory properties and a sketch of type inheritance are defined. Moreover, the schema thus defined is a PG. We hence expand the formal definition of PG Schema presented by [14] to introduce *edge cardinality*, *subtypes* and *supertypes* and the concept of *inheritance edge type*.

Definition 1. ***(Property Graph Schema)*** *A Property Graph Schema is a Property Graph Type, which is a triple* $(\mathcal{BT}, \mathcal{NT}, \mathcal{ET})$ *with* $\mathcal{BT}$ *a set of element types (defined in [14]),* $\mathcal{NT}$ *a set of node types,* $\mathcal{ET}$ *a set of edge types.*

- ***Subtypes****: A subtype is an element type such that it inherits from another element type—called the* ***supertype****.*
- ***Node Type****: A node type is a pair* (b, H)*, with* $b \in \mathcal{BT}$*,* $H \subseteq \mathcal{BT}$ *the set of* ***supertypes*** b *inherit from.*
- ***Edge Type****: the disjoint union of ordinary edge and inheritance edge types.*
- ***Ordinary Edge Type****: An ordinary edge type is a 4-uple* (s, e, t, c)*, with* $s \in \mathcal{NT}$ *the source node,* $t \in \mathcal{NT}$ *the target node,* $e \in \mathcal{BT}$*,* $c = ((i,k),(j,l)) \in (\{0,1\} \times \{1,N\})^2$ *the* ***cardinality****.*
- ***Inheritance Edge Type****: An inheritance edge type is a triple* (s, e, t)*, where* $s = (b, H) \in \mathcal{NT}$*,* $t \in H$*,* $e \in \mathcal{BT}$ *with a label that we denote "SubtypeOf". Inheritance edge types do not have any cardinality.*

In the remainder of the paper, unless stated otherwise, *edge type* refers to *ordinary edge type*. At last, let us formally define overlapping types:

Definition 2. ***(Overlapping Type)*** *An overlapping type is an element type such that it is a subtype of two or more supertypes.*

IV. A Method to Infer a PG Schema

We present here our method to infer a PG schema, from an input PG stored in Neo4j. The output schema is stored in a JSON file that can be loaded into Neo4j for visual inspection. We assume all nodes and edges in the PG are labeled and nodes are of the same type if and only if they have the same set of labels and edges are of the same type if and only if they have the same set of source node, target node and arc labels. These assumptions may be too strong in some cases. We will later present an alternative approach that deals with some of its shortcomings.

Our PG schema inference pipeline can be divided into three steps (cf. Fig. 1): Preprocessing (Section IV-A), Step 2: Types and Data Types Inference (MapReduce) (Section IV-B) and Node Hierarchy Inference (Section IV-C).

A. Step 1: Preprocessing

To begin with, the input PG, stored in Neo4j, needs to be serialized into JSON—the format required by the MapReduce algorithm. To this end, the graph is queried to match nodes and edges. To infer edge cardinality constraints, statistics are also collected. Then, the matched nodes and edges are serialized in such a way that proper node and edge type inference is guaranteed. From there, they are stored in `jsonline` files that will be input into the next step.

1) Preprocessing Queries: First, nodes are matched according to their labels via Cypher queries using the `neo4j` Python Driver library. Similarly, edges are matched according to their source, target node and arc labels.

Node and edge types such that none of their instances have properties are stored separately. Indeed, there is no need to infer their property data types. Hence, the MapReduce step can be skipped in these cases.

2) Edge Cardinalities: Edge cardinality constraints are then inferred using rules comparing the number of instances of the source nodes, target nodes and a given edge type. These are collected by the preprocessing queries. Let us denote this edge type $E = (s, e, t, c)$, with $s, t \in \mathcal{NT}$, $e \in \mathcal{BT}$ and c the cardinality. For instance, if there are more target nodes than source nodes and there are as many target nodes as edges of type E, then the cardinality of E is *one-to-many*. This means that an instance of the source node type s can be linked to many instances of the target node type t via the edge type E but that an instance of t can only be linked to one instance of s via E. These rules can be similarly declined for *one-to-one*, *many-to-one* and *many-to-many* relationships. The cardinality constraints can be further refined to take into account optional relationships. For the given edge type E, if there are fewer instances of source nodes than of its corresponding node type, s, then this edge type is *optional* for the source node type s (otherwise, the edge type is *mandatory* for s). This means that there may be nodes of type s that are not linked to nodes of type t via an edge of type E. The same rule can be applied for target nodes.

We used the following convention to encode cardinality constraints corresponding to each endpoint of a given edge:

- 1: One and only one relationship is allowed for the source (or target) node (it is mandatory)
- 1..*: One or more (mandatory)
- 0..1: Zero or one (optional)
- 0..*: Zero, one or more (optional)

The pseudocode given in Algorithm 1 showcases all possible combinations and how the corresponding edge cardinalities are identified.

In our pipeline, the cardinality constraint information is stored as an edge property with the key `meta_cardinality` and a `string` data type (e.g., `meta_cardinality` : "0..1:1..*" encodes a *one-to-many* relationship with the edge type being *optional* for the source node type and *mandatory* for the target node type). All edge types are subsequently stored in a Python `dict` and collected in a Python `list`.

3) Serialization to JSON: The Cypher queries output node and edge `neo4j` objects where property values are sometimes

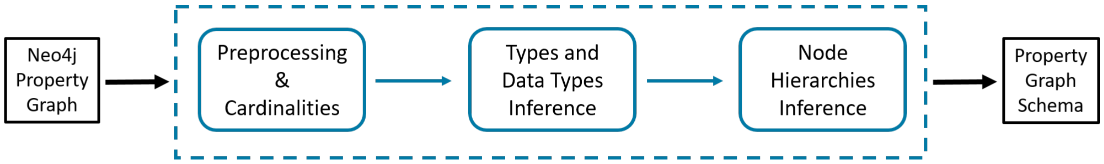

Neo4j Property Graph
Preprocessing & Cardinalities
Types and Data Types Inference
Node Hierarchies Inference
Property Graph Schema

Fig. 1: Schema inference main steps.

Algorithm 1: Edge Cardinality Constraints Inference for a Given Instance of the Edge Type $E = (s, e, t, c)$

```
▷ check whether the relationship is **mandatory** for the **source** node
**if** $nbSource < nbType\_s$ **then**
    $mandatorySource$ = False
**else**
    $mandatorySource$ = True
▷ check whether the relationship is **mandatory** for the **target** node
**if** $nbTarget < nbType\_t$ **then**
    $mandatoryTarget$ = False
**else**
    $mandatoryTarget$ = True
▷ check **cardinality** and append the edge dictionary accordingly
**if** $nbSource == nbTarget == nbEdges$ **then**
    cardinality is **one-to-one**
**else if** $nbSource > nbTargetandnbEdges == nbSource$ **then**
    cardinality is **many-to-one**
**else if** $nbSource < nbTargetandnbEdges == nbTarget$ **then**
    cardinality is **one-to-many**
**else**
    cardinality is **many-to-many**
```

incorrectly stored (e.g., a dictionary as a `string`). They are identified and converted accordingly to ensure a correct data type inference. Nodes and edges are then converted to Python `dicts` that are stored in `jsonline` files in such a way that correct type inference by the MapReduce method is guaranteed (cf. Section IV-B). The nodes file contains a single dictionary where each key-value pair represents a node. The key corresponds to its labels, sorted in alphabetical order and separated by colons. The value is a dictionary storing the properties as key-value pairs. In a similar fashion, The edges file contains a single dictionary where each key-value pairs represent an edge, with the value a dictionary storing the properties as key-value pairs. This time, the key corresponds to its starting node labels, its own labels and its target node labels, all separated by colons. The three sets of labels are separated by two colons.

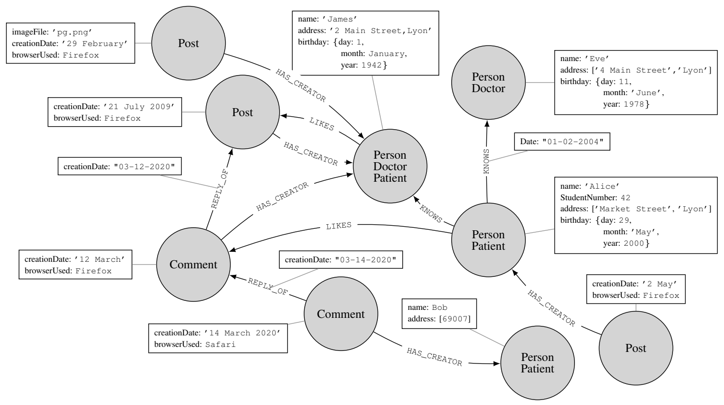


Fig. 2: Example PG instance, denoted G.

Example 1. *Let us consider a PG instance, denoted G, representing a social network of patients and doctors who can create and like posts and comments as wall as reply to them (cf. Fig. 2). This example is partially inspired from the LDBC Social Network Benchmark database [15]. Listings 1 and 2 showcase two example dictionaries encoding a node and edge instance, respectively. In Listing 2, the key "Patient:Person::KNOWS::Doctor:Person" corresponds to an edge type with the label `KNOWS` linking nodes with the label set {`Person`, `Patient`} to nodes with the label set {`Person`, `Doctor`}.*

```
{'Patient:Person': {
    'name': 'Alice',
    'birthday': {'day':29,
                 'month':'May',
                 'year':2000},
    'StudentNumber': 42,
    'address': ['Market Street','Lyon']}}
```

Listing 1: Dictionary storing a node instance

```
{'Patient:Person::KNOWS::Doctor:Person':
    {'date': '1993-06-02' }
}
```

Listing 2: Dictionary storing an edge instance

At the end of Step 1, we have two `jsonline` files ready to be input into the MapReduce algorithm, a list of node types with no properties and a list of all edge types—some of whom with missing properties (that will be determined in the MapReduce phase)—containing edge cardinality information.

B. Step 2: Types and Data Types Inference (MapReduce)

In this step, we aggregate nodes and edges by type and infer the property data types. The method used here was proposed by Baazizi et al. [4]. It can be summarized in two steps : i) a **Map** phase where all property values data types are inferred (cf. Listing 3) and ii) a **Reduce** phase where types are fused according to an equivalence relation. In our case, we use the *kind-equivalence* relation described in [4]. It fuses recursively types of the same *kind*, i.e. records with records, arrays with arrays and basic types with basic types. Basic types correspond to the `String`, `Number` and `Boolean` data types. The fusion of two basic types produces their union. The fusion of arrays outputs an array containing the fusion of their content. In the case of records, the different values of the two records are fused if and only if they share the same key. If one of the records contains keys that are not present in the other, then this particular key-value pair is deemed optional. Therefore, in our case, nodes will have their properties merged if and only if they share the same set of labels. Additionally, edges will have their properties merged if and only if they share the same set of source node, target node and arc labels. This complies with our assumption stated at the beginning of Section IV. The fusion function is recursively called so as to handle nested values (e.g., a record containing records).

Example 2. *Listing 3 and 4 illustrate the fusion of two* {`Person`, `Patient`} *node types using the kind-equivalence detailed above.*

```
{'Patient:Person': {
    'name': STRING,
    'birthday':{'day': NUMBER,
                'month': STRING,
                'year': NUMBER},
    'StudentNumber': NUMBER
    'address': [STRING] }}
{'Patient:Person': {
    'name': STRING,
    'address': [NUMBER] }}
```

Listing 3: Two JSON record types corresponding to two nodes present in G.

```
{'Patient:Person': {
    'name': STRING,
    'birthday':
        {'day': NUMBER,
        'month': STRING,
        'year': NUMBER,
        'meta_mandatory': FALSE},
    'address': [NUMBER + STRING],
    'StudentNumber': NUMBER ? }}
```

Listing 4: Human-readable fusion of the two JSON record types on the left-hand side using the *kind-equivalence*.

The output of the algorithm, which is a JSON record, is then parsed and stored in a human-readable Python `dict` where question marks are affixed to optional properties' data types (cf. Example 2). A "`meta_mandatory` : False" property is instead added to optional records. Next, the dictionary is merged with the node types with no properties and the list of edge types containing cardinality constraints information. Therefore, the output of this step is a preliminary PG schema of the input PG that is only missing node hierarchies information.

C. Step 3: Node Hierarchies Inference

The final step is to infer node type hierarchies, so as to obtain a schema satisfying Definition 1. Inferring edge hierarchies is unnecessary in Neo4j graphs, since edges can only be associated with a single label. Nevertheless, we would expect our hierarchy inference technique (outlined in Algorithm 2) to extend straightforwardly to edge hierarchies.

Algorithm 2: Node Hierarchy Inference (Labels-oriented variant)

```
1                                  ▷ Identify supertypes
2  supertypes = list of the pairwise intersections of the
     node label sets
3  for stype in supertypes do
4      add stype to nodes if needed
5                                  ▷ Identify subtypes
6  nodeLabels = list of node label sets
7  for $i \leftarrow 0$ to length(nodeLabels) $-1$ do
8      $nset0 = nodeLabels[i]$
9      for $j \leftarrow 0$ to length(nodeLabels$[i:]$) $-1$ do
10         ▷ Two given node types are compared only
           once
11         $nset1 = nodeLabels[j]$
12         if $nset0 \neq nset1$ then
13             if $nset0 \subset nset1$ then
14                 add $nset1$::SubtypeOf::$nset0$ edge
15             else if $nset1 \subset nset0$ then
16                 add $nset0$::SubtypeOf::$nset1$ edge
```

First, node **supertypes** corresponding to subsets of labels present in two or more node types label sets are inferred (l. 1-4). To this end, we take the pairwise intersection of the label sets of all node types inferred during Step 2. Some of them may not be instantiated in the input PG. Nonetheless, they can further reveal the node type hierarchy structure. For instance, let us consider a {`Person`, `Doctor`} and a {`Person`, `Patient`} node type. The node type labeled {`Person`} is thus identified as a *supertype*.

The second step is to identify all **subtypes** (l. 5-16). We assumed earlier that node types are characterized by their labels. We therefore consider a node type (with label set A) to be a *subtype* of a distinct node type (with label set B) if $B \subsetneq A$. This enables us to automatically handle overlapping types and hierarchies of arbitrary depths, as long as they are reflected in the labels. Thus, the label sets of all node types, including those inferred previously, are compared in pairwise fashion to identify subtypes. For example, {`Person`, `Patient`, `Doctor`} is a *subtype* of {`Person`, `Doctor`}. The corresponding inheritance edges are then created and added to the schema.

The time complexity of the node hierarchy inference is the square of the number of node types inferred in the previous step. As long as this is significantly smaller than the number of nodes in the input PG, we can consider the computational complexity of this step to be $\mathcal{O}(1)$.

With the node type hierarchy inferred, the PG schema is complete. It is stored in a JSON file using the format described in Section IV-A.

D. Method Variant: Labels as Properties

So far, we have assumed that all nodes in the input PG are labeled and that node labels characterize node types. However, these assumptions may sometimes be unsuitable. Indeed, some graphs may contain unlabeled nodes and information provided by the properties may be lost when only taking labels into account. For example, let us consider the PG G (cf. Section IV-A) and assume that the nodes labeled {`Person`, `Patient`} can be partitioned into two groups: those with a `StudentNumber` property key, corresponding to patients who are students, and those without one. With the previous approach, we had missed this subtlety and only identified a single {`Person`, `Patient`} node type with an optional `StudentNumber` key (cf. Listings 3 and 4 in Example 2 and Fig. 3). Therefore, we propose to consider labels as properties with a `Void` data type and to use property key sets (which now include labels) instead of label sets to characterize node types and thusly identify node types and hierarchies. To merge nodes, we hence use the *$\mathcal{L}$-driven reduction* [4], which fuses two records if and only if they share the same property key sets. As a result, no optional property can be inferred but rather property key co-occurrence information is identified. Moreover, unlabeled nodes can henceforth be considered on the same level as labeled nodes. In Neo4j, edges must have exactly one label. So, all the input PGs we considered contain single-labeled edges. We thus continue to utilize our labels-oriented approach to handle them. If the source node or target node is unlabeled, it is referred to by its property keys in place of labels. Our PG schema inference pipeline in this paradigm is very similar to the former and can be divided into the same three steps.

Example 3. *Fig. 3 and 4 depict the schemas inferred from the PG G (cf. Section IV-A) using, respectively, the labels- and properties-oriented variant. As discussed above, the first approach overlooks the information provided by the properties, resulting in missing node types. This is resolved with the properties-oriented variant, although spurious types are also inferred.*

V. Evaluation

Our method is implemented in Python 3 and is based on Neo4j 3.5. The graphs are queried with Cypher through the Neo4j Python driver. For the MapReduce step, we use—with the permission of the authors—the implementation developed by Baazizi et al. [4], which runs with Spark 2.4.5. All experiments were performed on an Openstack Virtual Machine with twelve 2GHz 64-bits Intel Xeon CPUs, 62 GB of memory and a 1.5 TB hard drive.

A. Datasets

The implementations of our schema inference method were tested on several datasets. Their characteristics are reported in Table I. We evaluated our schema inference method on the **LDBC Social Network Benchmark** (**LDBC**) [15], a synthetic social network available for benchmarking. It contains single-labeled nodes and a ground-truth schema incorporating node hierarchies is available. We also used two **Neuprint** datasets, corresponding to neuronal networks of different parts of the fruit fly brain: i) the mushroombody (**mb6**) [16] and ii) the medulla (**fib25**) [17]. Accompanied by a ground-truth schema, they contain multi-labeled nodes and a large diversity of property values data types, such as JSON records or `neo4j` cartesian 3D points. We tested as well our pipeline on a **Covid-19 knowledge graph** (**covid19**), available as a public Neo4j graph (https://covidgraph.org/). It compiles information about knowledge on Covid-19, and more broadly coronaviruses. This graph is being assembled by the CovidGraph project, which is currently on-going. The graph is continuously evolving and hence so are the corresponding schemas. The results presented in this paper are those obtained with the April 2020 version, which notably holds multi-labeled nodes and five unlabeled nodes.

B. Metrics

We have access to a ground truth schema for the LDBC and Neuprint datasets. We can thus compute the precision, recall and F1-score of the node and edge types to assess the quality of our inference. We consider an inferred type that is present in the ground-truth schema as a True Positive (TP). An inferred type that does not exist in the ground-truth schema is considered as a False Positive (FP). A type that is present in the ground-truth but not in the inferred schema is considered as a False Negative (FN). Precision accounts for the proportion of identified types that were actually in the ground-truth schema, while the recall provides the proportion of ground-truth types that were inferred. The F1-score gives an average of the precision and recall that can be used as an indication to balance these two metrics. Let us now recall the definition of these commonly used metrics:

$$Precision = \frac{TP}{TP + FP} \tag{1}$$

$$Recall = \frac{TP}{TP + FN} \tag{2}$$

$$F1 = \frac{2(Precision * Recall)}{Precision + Recall} \tag{3}$$

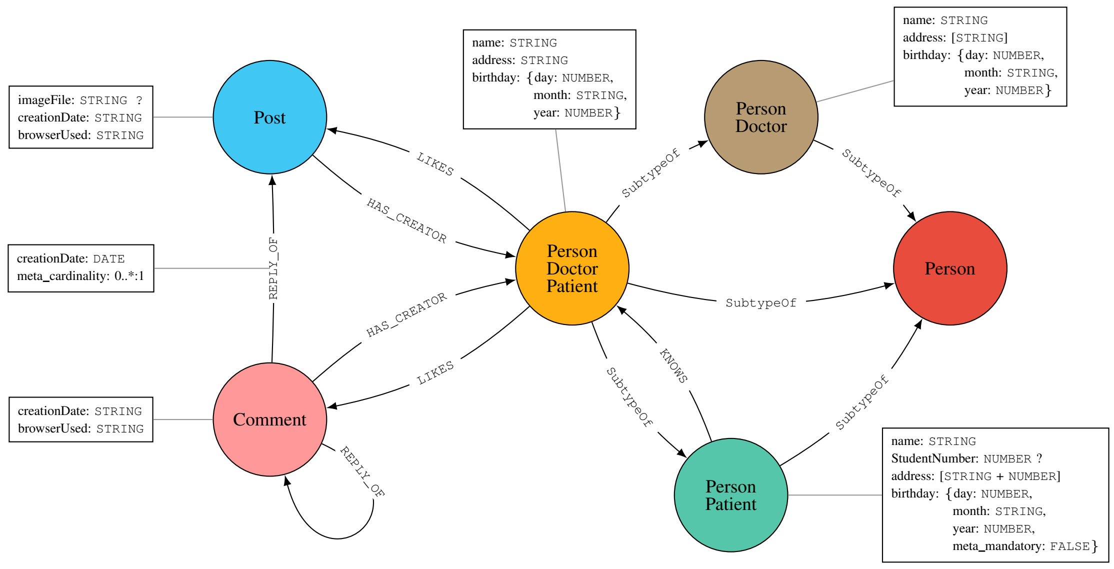


Fig. 3: PG schema inferred from G, a toy example PG, using our labels-oriented variant. To improve visibility, some edges and edge cardinality information are not represented (e.g., (`Patient`,`Person`)-[`KNOWS`]->(`Patient`,`Doctor`)). `Person` is a supertype identified through label sets. The `Message` supertype of the `Post` and `Comment` nodes could not be inferred, nor could the {`Person`, `Patient`} type be split into `Student` and non-`Student` types.

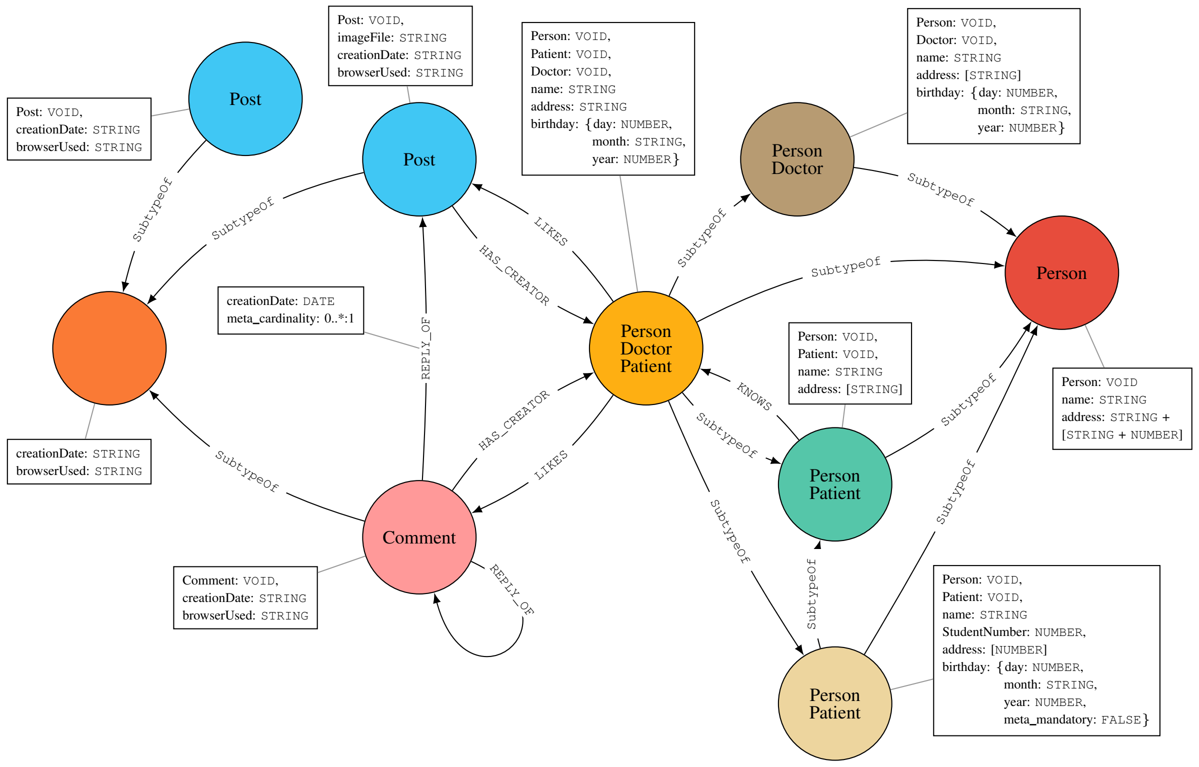


Fig. 4: PG schema inferred from G, a toy example PG, using our properties-oriented variant. To improve visibility, some edges and edge cardinality information are not represented. A few node types that could not be inferred with the labels-oriented approach were identified thanks to their property key sets: the unlabeled node corresponds to a `Message` supertype of the `Post` and `Comment` node types; the {`Person`, `Patient`} node was split into `Student` and non-`Student` type. The `Person` supertype, also inferred using the other variant, was identified. As a side effect, a spurious `Post` node type was inferred.

TABLE I: Characteristics of the different datasets used for evaluation.

Dataset Name	Number of Nodes	Number of Edges	Number of Node Labels	Number of Edge Labels	Number of Unlabeled Nodes	Nested or Multiple Values
mb6	486,267	961,571	10	3	0	Yes
fib25	802,479	1,625,439	10	3	0	Yes
covid19	10,447,251	25,340,047	60	73	5	Yes
LDBC	1,577,397	8,179,418	7	14	0	No

TABLE II: Information about the schema inferred from different datasets with our labels-oriented variant.

Dataset Name	Number of Node Types	Number of Edges Types	Number of Inheritance Edges Types	Max Node Hierarchies Depth	Overlapping Types
mb6	5	10	1	1	No
fib25	5	10	1	1	No
covid19	77	159	43	1	Yes
LDBC	7	21	0	0	No

TABLE III: Information about the schema inferred from different datasets with our properties-oriented variant.

Dataset Name	Number of Node Types	Number of Edges Types	Number of Inheritance Edges Types	Max Node Hierarchies Depth	Overlapping Types
mb6	68	795	786	9	Yes
fib25	47	427	418	8	Yes
LDBC	17	72	51	5	Yes

TABLE IV: Precision, recall and F1-score of the node and edge types.

	Labels-oriented						Properties-oriented					
	Precision		Recall		F1		Precision		Recall		F1	
Dataset	nodes	edges	nodes	edges	nodes	edges	nodes	edges	nodes	edges	nodes	edges
mb6	0.80	0.83	0.80	0.83	0.80	0.83	0.29	0.01	0.80	0.83	0.43	0.01
fib25	0.80	0.83	0.80	0.83	0.80	0.83	0.09	0.01	0.80	0.83	0.15	0.27
ldbc	1.00	1.00	0.54	0.70	0.70	0.82	0.47	0.47	0.62	0.75	0.53	0.58

C. Results and Discussion

In this section, we present and discuss the results of our evaluation. Table II and III display, for both variants, different information about the inferred schemas, such as the number of node and edge types. In this section, edge type refers to the union of ordinary and inheritance edge types (cf. Definition 1).

1) Quality: In both the Neuprint and LDBC datasets, the property value data types, as well as the edge cardinality constraints of the correctly identified edge types, have been inferred accurately. The precision, recall and F1-score of the node and edge types have been recorded in Table IV and demonstrate the overall good quality of the types inferred with our labels-oriented approach.

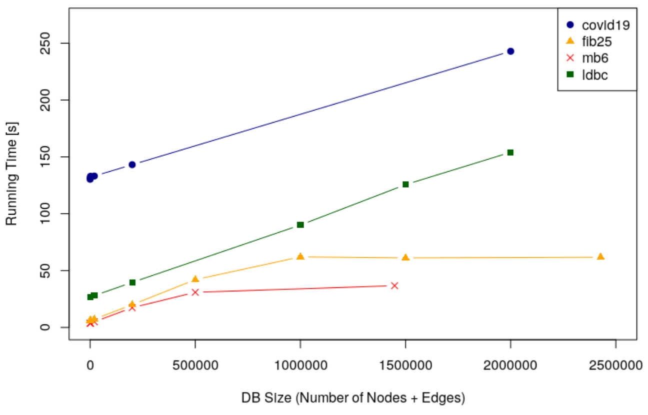
a) Labels-Oriented Approach: All three metrics—notably the precision, which ranges from 0.80 to 1.00—are reasonably high. We can also note that the different metrics are identical for both Neuprint datasets (mb6 and fib25). Indeed, they share both the ground-truth and inferred types. It is worth mentioning that only one node type and its corresponding inheritance edge type absent in the ground-truth schema have been identified. This is due to an inconsistency in the labeling of this particular node type where some of its instances have more labels than others. This caused our algorithm to incorrectly aggregate them into distinct node types. This highlights the reliance of our method on the proper labeling of the graph entities and hence its sensitivity to noisy labels. A statistical approach to the type inference, such as clustering methods (like the one in [8]), would allow to group together nodes (or edges) that share similar, but not identical, label sets. Combining clustering with graph embedding [18], which maps the input graph to a low-dimension continuous space while preserving the inherent characteristics of the graph to the best possible extent, like in [19], could also emerge as a promising solution. Data types and node hierarchies would still need to be inferred, possibly by integrating such approaches with our schema inference method.

In the LDBC graph, all inferred types exist in the ground-truth schema. However, none of the ground-truth hierarchies were discovered. Indeed, they were either defined via a `type` property, instead of labels, or identifiable only through properties in common. The former might be addressed with a semantic approach, while the latter is partially overcome with our properties-oriented variant.

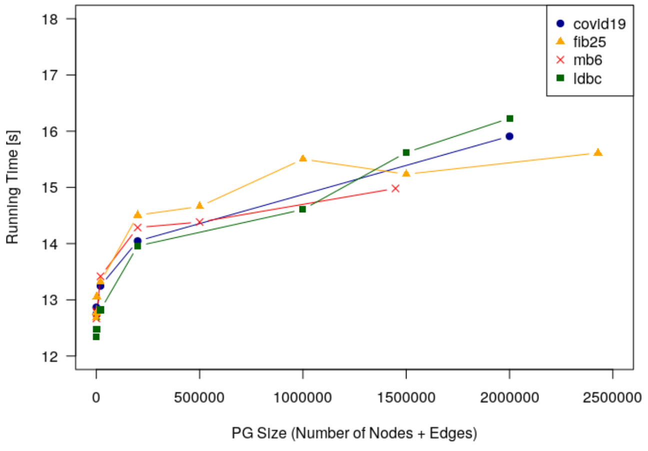
b) Properties-Oriented Approach: The low precision and F1 scores obtained with the properties-oriented approach can be explained by the inference of numerous spurious types—in addition to the correct ones. For instance, in the mushroombody dataset (mb6) 63 additional node types were inferred. Indeed, since we are considering property sets to infer node types, for a given label set, we infer as many node types as there are combinations of properties—although in some cases, this behavior is expected (see the example in Section IV-D). Furthermore, different node types may have common property keys even if they are not subtypes of a common supertype. For example, in the LDBC graph, both the node types `Place` and `Person` hold a property with the key `name`. Nonetheless, in the Neuprint graphs, the number of True Positives and False Negatives remain unchanged from one variant to the other. The recall remains thus constant. Even more remarkably, in the LDBC graph, more types present in the ground-truth are identified than with the labels-oriented variant. These types correspond to supertypes that could not be inferred using solely the node labels. This is reflected in the recall scores, which increase from 0.54 to 0.62, for nodes, and from 0.70 to 0.75, for edges, when switching from the labels-oriented variant to the properties-oriented one.

c) Labels-Oriented vs Properties-Oriented Approaches: To summarize, the labels-oriented variant outputs schemas with a very good precision. However, it is missing node types that can only be inferred through properties-related information. This is partially overcome by the properties-oriented variant, which is marked by an improved recall. Nonetheless, since many spurious types are inferred as well, this is done at the expense of the precision. Hence, the labels-oriented variant should be preferred, either when the node hierarchies in the input PG are defined through labels, or when there are no hierarchies—such as in the Neuprint graphs. On the other hand, the properties-oriented variant should be picked when properties are crucial to the inference process, such as in the presence of unlabeled nodes or when hierarchies are determined by property co-occurrence information. Combining the outputs of both approaches to only capture the wanted node types, while beyond the scope of this paper, is interesting as a topic of future investigation.

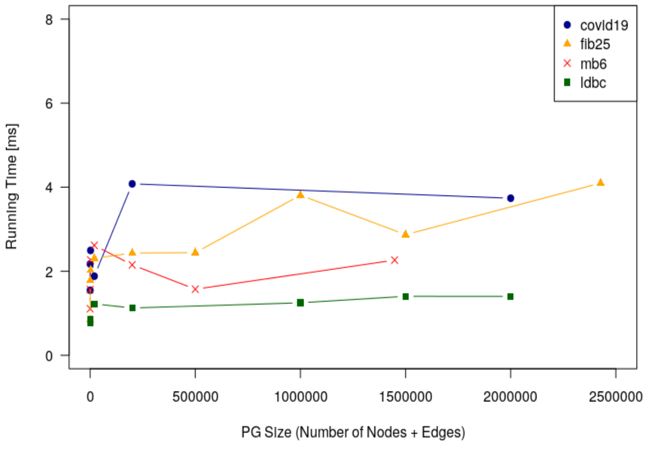
2) Scalability: We obtained the average running times of our schema inference pipeline for portions of different sizes of the datasets. The times discussed in this section were acquired with our labels-oriented implementation. Those from our properties-oriented implementation are of the same order of magnitude. The first step (cf. Fig. 5a), where we match every single node and edge of the input PG, brings to light the problem of the overhead caused by the Cypher queries, which increases with the size of the input PG. Indeed, the running times can go up to about 1900s for the complete covid19 dataset, with its 10M nodes and 25M edges (this data point is not represented in Fig. 5a so the running times of the other datasets, which are much smaller, can be visible). As such, the pipeline running time is dominated by this step. Still, it seems that it is at worst linear in the input size. Fig. 5b displays the sublinear behavior of the running times of Step 2, which is as expected with regards to [4]. Moreover, our parsing function has an average running time smaller than 2ms, which is satisfactory. On average, Step 3 (cf. Fig. 5c) runs in less than 5ms and is constant when the input PG size increases, which comforts our complexity analysis



(a) Step 1: Preprocessing.



(b) Step 2: Types and data types inference (MapReduce).



(c) Step 3: Node hierarchy inference.

Fig. 5: Average running times for different datasets when the input PG size varies. Step 1 brings to light the problem of the Cypher queries overhead, while step 2 and 3 scale well.

carried out in Section IV-C. Additionally, Step 2 and 3 are not sensitive to the heterogeneity in the complexity of data types and structures displayed in the different datasets.

VI. Conclusion and Future Work

To conclude, we have presented an end-to-end PG schema inference method. Our three-step pipeline leverages a MapReduce type inference method and introduces a novel node hierarchy inference technique to output PG schemas that handle complex and nexted property values, multi-labeled nodes, node hierarchies and overlapping node types, in addition to containing information about edge cardinality constraints and optionality of properties.

We have presented and empirically evaluated two implementations that both scale well, if one sets aside the preprocessing step which relies on the performances of the Neo4j Cypher queries engine. The *labels-oriented* variant provides an inferred schema of good quality with high precision, recall and F1 scores. Its main shortcomings are its sensitivity to the graph labeling and the loss of property co-occurrence information that could lead to additional supertype identification. This is resolved by our *properties-oriented* approach, which concurrently improves recall scores. However, in the process, many extraneous types are inferred.

A solution worth exploring in future work would be to find a way to combine the outputs of these two variants to exclusively retain the wanted node types. Our schema inference method is also sensitive to variation in the labels and property keys, be it due to inconsistencies or multilingual naming. To overcome this, a statistical approach to the type inference step, such as clustering on the nodes and edges of the input instances, possibly combined with graph embedding, which can take into account semantic information to simplify graph representations, could be considered.

References

[1] A. Bonifati, G. Fletcher, H. Voigt, and N. Yakovets, *Querying Graphs*. Morgan & Claypool Publishers, 2018, vol. 10.

[2] R. Harmer, Y. Le Cornec, S. Légaré, and E. Oshurko, "Bio-curation for cellular signalling: The kami project," *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, vol. 16, no. 5, pp. 1562–1573, 2019.

[3] P. W. Battaglia and et al., "Relational inductive biases, deep learning, and graph networks," *ArXiv*, vol. abs/1806.01261, 2018.

[4] A. Baazizi, D. Colazzo, G. Ghelli, and C. Sartiani, "Parametric schema inference for massive JSON datasets," *The VLDB Journal*, vol. 28, no. 4, 2019.

[5] O. Hartig and J. Hidders, "Defining schemas for property graphs by using the GraphQL schema definition language," in *Proceedings of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*. Association for Computing Machinery, Jun 2019.

[6] W. Lyon, "Inferring graphql type definitions from an existing neo4j database," 2019. [Online]. Available: https://blog.grandstack.io/inferring-graphql-type-definitions-from-an-existing-neo4j-database-dadca2138b25

[7] S. Normey Gómez, L. Etcheverry, A. Marotta, S. Normey, and M. P. Consens, "Findings from Two Decades of Research on Schema Discovery using a Systematic Literature Review," in *AMW*, 2018.

[8] A. Lutov, S. Roshankish, M. Khayati, and P. Cudré-Mauroux, "StaTIX - Statistical Type Inference on Linked Data," in *Proceedings - 2018 IEEE International Conference on Big Data, Big Data 2018*. Institute of Electrical and Electronics Engineers Inc., Jan 2019, pp. 2253–2262.

[9] R. Bouhamoum, K. Kellou-Menouer, Z. Kedad, and S. Lopes, "Scaling Up Schema Discovery for RDF Datasets," in *2018 IEEE 34th International Conference on Data Engineering Workshops (ICDEW)*, 2018, pp. 84–89.

[10] V. D. Blondel, J.-L. Guillaume, R. Lambiotte, and E. Lefebvre, "Fast unfolding of communities in large networks," *Journal of Statistical Mechanics: Theory and Experiment*, vol. 2008, no. 10, p. P10008, Oct 2008.

[11] R. Angles, "The Property Graph Database Model," *Proceedings of the 12th Alberto Mendelzon International Workshop on Foundations of Data Management*, vol. 2100, 2018. [Online]. Available: http://ceur-ws.org/Vol-2100/paper26.pdf

[12] M. Wu, "Property Graph Type System and Data Definition Language," Oct 2018. [Online]. Available: http://arxiv.org/abs/1810.08755

[13] P. Atzeni, S. Ceri, S. Paraboschi, and R. Torlone, *Database Systems concepts, languages and architectures*. McGraw-Hill Publishing Company, 1999.

[14] A. Bonifati, P. Furniss, A. Green, R. Harmer, E. Oshurko, and H. Voigt, "Schema validation and evolution for graph databases," in *Conceptual Modeling*. Cham: Springer International Publishing, 2019, pp. 448–456.

[15] LDBC Social Network Benchmark task force, "The LDBC Social Network Benchmark (version 0.3.2)," Linked Data Benchmark Council, Tech. Rep., 2019. [Online]. Available: https://ldbc.github.io/ldbc_snb_docs/ldbc-snb-specification.pdf

[16] S.-y. Takemura and et al., "A connectome of a learning and memory center in the adult Drosophila brain," *eLife*, vol. 6, Jul 2017.

[17] ——, "Synaptic circuits and their variations within different columns in the visual system of Drosophila," *Proceedings of the National Academy of Sciences*, vol. 112, no. 44, pp. 13 711–13 716, Nov 2015.

[18] Q. Wang, Z. Mao, B. Wang, and L. Guo, "Knowledge graph embedding: A survey of approaches and applications," *IEEE Transactions on Knowledge and Data Engineering*, vol. 29, no. 12, pp. 2724–2743, 2017.

[19] B. Rozemberczki, R. Davies, R. Sarkar, and C. Sutton, "Gemsec: Graph embedding with self clustering," *2019 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining (ASONAM)*, pp. 65–72, 2019.