CONTENTS

Schema Inference for Property Graphs

Hanâ Lbath
Lyon 1 University, ENS Lyon
France
hana.lbath@ens-lyon.fr

Angela Bonifati
Lyon 1 University
France
angela.bonifati@univ-lyon1.fr

Russ Harmer
CNRS, ENS Lyon
France
russ.harmer@ens-lyon.fr

Abstract

Graphs are pervasive in many applications in which interconnected data are used to represent, explore and predict digital and real-world phenomena. Oftentimes, graph data comes without a predefined structure and in a constraint-less fashion, thus leading to inconsistency and poor quality. 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 obtained during the first step. The third step analyzes subtypes and supertypes to infer node hierarchies. We present our schema inference pipeline under two variants, namely a label- and a property-oriented variant. Finally, we experimentally evaluate and compare its scalability and accuracy on several real-life datasets. To the best of our knowledge, our work is the first to address schema inference for property graphs.

1 Introduction

Over the past decade, graph-based knowledge representation has been becoming increasingly popular, be it with the advent of graph databases [4] as an alternative to relational databases, or to model complex systems, from social and fraud detection networks to smart city grids or neuronal networks. Graphs are applicable to all settings in which interconnected data are used to represent, explore and predict digital and real phenomena. Understanding the connections in the data is fundamental for companies to carry out analytical and machine learning tasks.

Oftentimes, graph data comes without a structure and in a constraint-less fashion, thus leading to inconsistency and poor quality. According to a recent survey [12], the most recurrent task for users of real-world graphs is data integration. As a matter of fact, graphs naturally lend themselves to information reconciliation and integration. However, the integration of large-scale graphs, e.g. in knowledge bases such as DBPedia, Wikipedia or the more recent CovidGraph, might turn out to be incorrect and error-prone if not guided by means of schema constraints. These constraints are also important pillars for query optimization and metadata management, the latter being unexplored topics in graph databases. There are several models for representing graphs. Among the most popular is the property graph (PG) data model, which is a multigraph with both labeled nodes and edges, along with property value pairs associated to both. It has gained adoption with systems such as ArangoDB, HANA Graph, Neo4j, Oracle PGX, TigerGraph, Titan, etc.

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 has to be stabilized as in many data management settings. 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 based on types as 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 non-trivial and challenging task. Existing PG schema inference methods in available graph databases (such as Neo4j) are simplistic in that they can only output basic edge types and node types and do not take into account the complexity of the PG data model. In particular, they cannot handle complex data types, overlapping node types or node hierarchies. A principled approach to PG schema inference is currently missing and highly desirable given the interest in an ongoing ISO SC32/ WG3 standardization process of PG schemas, involving people from academia and industry in the LDBC community[1].

In this work, we address this problem and present a novel end-to-end schema inference method covering the entire spectrum of features of the PG data model. Our method tackles complex and nested property values, multi-labeled and unlabeled nodes, node hierarchies and overlapping node types as well as edge cardinality constraints and optionality of properties. We also introduce two variants of our method, a label-oriented and a property-oriented one and investigate their pros and cons.

To enable scalability, our method leverages a MapReduce approach [2] developed for schema inference from JSON datasets, which both aggregates types and identifies data types. However, the JSON data model and the PG data model are significantly different. While the former can be seen as an edge-labeled tree-structured data model, the latter is more expressive as it is a multi-graph with novel schema components such as subtyping and edge cardinality constraints. Because of that, schema inference for property graph is a challenging and non-trivial problem that we tackle in this paper for the first time.

The schema inference pipeline we designed[2] can be divided into three main steps. First, we employ Cypher queries to extract and serialize the nodes and edges of the input PG, in addition to gathering information needed to infer edge cardinality constraints. Cypher[3] is an open-source graph query language developed by Neo4j, inspired from SQL and adopted by several graph database vendors. 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

[1] More information can be found at: https://www.gqlstandards.org
[2] The source code is available at: https://gitlab.com/Hgit/pgsinference
[3] http://www.opencypher.org

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 datasets.

2 Related Work

Several vendors propose graph databases supporting the PG data model, such as Neo4j, Oracle PGX, TigerGraph, RedisGraph and GraphQL. Neo4j offers a limited possibility to view a schema of the database via a Cypher query that outputs a single-labeled directed graph displaying which types of nodes can be connected together and through which types of edges. However, there is a lack of support of the PG data model in its full potential [4]. In particular, 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, which is a crucial capability of the PG data model. Furthermore, properties, edge cardinality constraints and node type hierarchies are disregarded. Nonetheless, other Cypher queries output 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 [7] 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 [10]. Typical examples are clustering techniques, like [9] and [6], which infer *types* in RDF datasets. Both are based on the assumption that the more *properties* two *entities* share, the likelier they belong to the same *type*. To this end, they group entities according to a similarity metric. They also both handle hierarchical and overlapping types. In [6], a density-based clustering method, DBSCAN, is adopted. In [9] a faster and more accurate clustering method, called StaTIX, is proposed. It uses the cosine similarity metric and is based on community detection. However, none of these techniques are suitable for PGs.

In [2], 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 [2] does not tackle explicitly the problem of overlapping types. Due to the remarkable differences between the JSON data model and the PG data model, their method is not directly applicable to PGs.

In summary, none of the above approaches fully satisfy our criteria for PG schema inference: inference of types, basic and complex data types, overlapping types and node hierarchies, and, to the best of our knowledge, ours is the first work presenting a schema inference method specifically tailored to PGs.

3 Preliminaries

In this section, we recall the definition of property graphs [1, 4, 5] and extend a PG schema definition to best fit the schema inference process presented in this paper.

Let $\mathcal{O}$ be a set of *objects*, $\mathcal{L}$ be a finite set of *labels*, $\mathcal{K}$ be a set of *property keys*, and $\mathcal{N}$ be a set of *values*. We assume these sets to be pairwise disjoint.

Definition 3.1. A *property graph* is a structure $(V, E, \eta, \lambda, \nu)$ where

- $V \subseteq \mathcal{O}$ is a finite set of objects, called vertices;
- $E \subseteq \mathcal{O}$ is a finite set of objects, called edges;
- $\eta : E \rightarrow V \times V$ is a function assigning to each edge an ordered pair of vertices;
- $\lambda : V \cup E \rightarrow \mathcal{P}(\mathcal{L})$ is a function assigning to each object a finite set of labels (i.e., $\mathcal{P}(S)$ denotes the set of finite subsets of set S); and,
- $\nu : (V \cup E) \times \mathcal{K} \rightarrow \mathcal{N}$ is a partial function assigning values for properties to objects;

such that $V \cap E = \emptyset$ and the domain of ν is finite.

In [5], a Data Definition Language (DDL) for PGs is proposed where the obtained schema is a PG itself. We expand it to introduce *edge cardinality*, *subtypes* and *supertypes* and the concept of *inheritance edge type*. All these are key concepts to make our inferred PG schemas as expressive and accurate as possible.

Definition 3.2. (**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, $\mathcal{NT}$ a set of *node types*, $\mathcal{ET}$ a set of *edge types*.

- **Property type**: A property type is a pair (k, t), where $k \in \mathcal{K}$ is the property key and $t \in \mathcal{T}$ is its data type.
- **Element type**: An element type $b \in \mathcal{BT}$ is a quadruple (l, P, M, E), where $l \in \mathcal{L}$ is a label, P is a set of property types, $M \subseteq P$ is a subset of mandatory property types and $E \subseteq \mathcal{BT}$ is the set of element types that b extends.
- **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.
- **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.
- **Ordinary Edge Type**: An ordinary edge type is a quadruple (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**.
- **Edge Type**: the disjoint union of ordinary edge and inheritance edge types.

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

Definition 3.3. (**Overlapping Type**) An overlapping type is an element type which is a subtype of two or more supertypes.

4 Inferring Property Graph Schemas

In this section we present our method to infer a PG schema. We assume all nodes and edges in the PG are labeled. Nodes are of the same type if and only if they have the same set of labels, while edges are of the same type if and only if they share their set of source nodes, target nodes and edge labels. These assumptions may be too strong in some cases. We will present in Section 4.4 an alternative that deals with some of its shortcomings.

Our PG schema inference pipeline can be divided into three main steps (cf. Fig. 1).

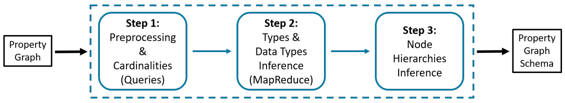

Property Graph
Step 1: Preprocessing & Cardinalities (Queries)
Step 2: Types & Data Types Inference (MapReduce)
Step 3: Node Hierarchies Inference
Property Graph Schema

Figure 1: Schema inference main steps.

4.1 Step 1: Preprocessing

To begin with, the input PG 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.

4.1.1 Preprocessing Queries. First, nodes are matched according to their labels via Cypher queries. Similarly, edges are matched according to their source, target node and edge labels. Node and edge types such that none of their instances have properties are stored separately. As there is no need to infer their property data types, the MapReduce step can be skipped in these cases and they can be directly processed by the third step.

4.1.2 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.

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 *optional* for the source node type and *mandatory* for the target node type).

4.1.3 Serialization to JSON. The Cypher queries output node and edge `neo4j` objects where property values are sometimes 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 dictionaries that are stored in `jsonline` files in such a way that correct type inference by the MapReduce method is guaranteed (cf. Section 4.2). 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. Similarly, the edges file contains a single dictionary where each key-value pair represents 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.

4.1.4 Example. Let us set a PG instance G of a social network of patients and doctors who can create and like posts and comments as well as reply to them. This is partially inspired by the LDBC Social Network Benchmark database [8]. Listings 1 and 2 are dictionaries encoding a node and edge instance.

```
{'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 containing edge cardinality information but no properties (they will be added in Step 2).

4.2 Step 2: Types and Data Types Inference (MapReduce)

In this step, we aggregate nodes and edges by type and infer the property data types by relying on MapReduce [2]. It can be summarized in two steps : i) a **Map** phase where all property value data types are inferred (cf. Listing 3) and ii) a **Reduce** phase where types are fused according to an equivalence relation. Here, we use the *kind-equivalence* relation described in [2] (cf. Listing 4). It fuses recursively types of the same *kind*, i.e. records with records, arrays with arrays and basic types (`String`, `Number` and `Boolean`) with basic 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, 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 4. The fusion function is recursively called so as to handle nested values.

The output of the algorithm, which is a JSON record, is then parsed and stored in a human-readable dictionary where question marks are affixed to optional properties' data types. A property "`meta_mandatory` : False" 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. Thus, the output of this step is a preliminary PG schema of the input PG, but it is still missing subtyping information.

4.2.1 Example. Listing 3 and 4 illustrate the fusion of two {`Person`, `Patient`} nodes 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: Fusion of the two JSON record types on the left-hand side using the *kind-equivalence*.

4.3 Step 3: Inference of Node Hierarchies

The final step is to infer node type hierarchies so as to obtain a schema satisfying Definition 3.2. Inferring edge hierarchies is unnecessary in Neo4j graphs, since edges can only be associated with a single label. Nevertheless, we expect our hierarchy inference technique to straightforwardly extend to edge hierarchies.

Algorithm 1: Node Hierarchy Inference (Label-Oriented Variant)

```
   input  : schema nodes and edges dictionaries from Step 2
   output : updated schema nodes and edges dictionaries and schema file
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 type 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. 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 thus 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 quadratic in the number of node types inferred in the previous steps. However, since the number of node types is typically smaller than the size of the PG, this complexity remains bearable in practice, as also shown by our evaluation.

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

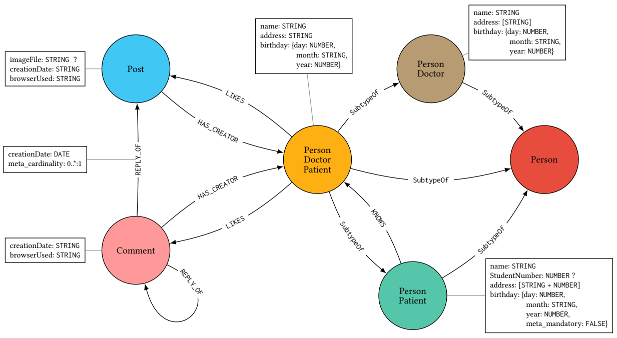


Figure 2: An excerpt of the PG schema inferred from the PG G using our label-oriented variant.

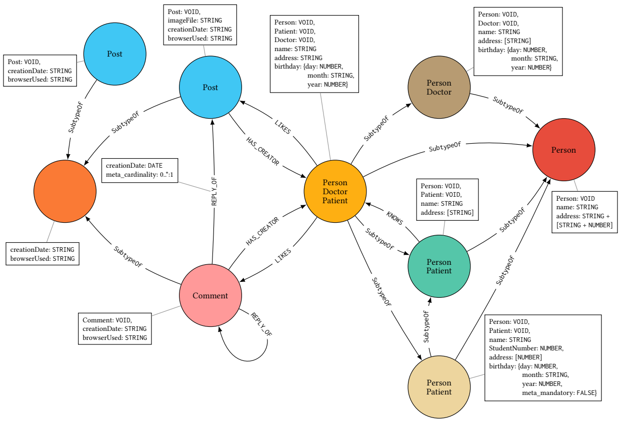


Figure 3: An excerpt of the PG schema inferred from the PG G using our property-oriented variant.

4.4 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. Example 4.1.4) 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 and Fig. 2). 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 thus identify node types and hierarchies. To merge nodes, we hence use *$\mathcal{L}$-driven reduction* [2], 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 label-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.

4.4.1 Example. Fig. 2 and 3 depict the schemas inferred from the PG *G* using both variants. As discussed above, the first approach overlooks the information provided by the properties, resulting in missing node types (e.g., a supertype of the `Post` and `Comment` nodes could not be inferred). This is resolved with the property-oriented variant where an unlabeled supertype could be inferred (the unlabeled orange node in Fig. 3).

5 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. The MapReduce step is based on the implementation in [2], 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.

5.1 Datasets and Metrics

We have used several datasets in our experimental study (cf. Table 1). We evaluated our schema inference method on the **LDBC Social Network Benchmark** (**LDBC**) [8], a synthetic social network. It contains single-labeled nodes and comes equipped with a ground truth schema incorporating node hierarchies. We also used two **Neuprint** datasets, corresponding to neuronal networks of different parts of the fruit fly brain: i) the mushroombody (**mb6**) [14] and ii) the medulla (**fib25**) [13]. Accompanied by a ground truth schema, they contain multi-labeled nodes (as opposed to LDBC) and a large diversity of property value data types, such as JSON records or `neo4j` cartesian 3D points. We tested as well our pipeline on a **Covid-19 graph** (**covid19**) (https://covidgraph.org/). This graph is being assembled by the CovidGraph project, which is currently ongoing. 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. No ground truth is available for the schema of the latter graph.

To assess the quality of our schema inference, we have used the precision, recall and F1-score of the node types and edge types. We consider an inferred type that is (not) present in the ground truth schema as a True Positive (TP) (False Positive (FP), respectively). 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 are present in the ground truth, while the recall gives the proportion of ground truth types that were inferred. The F1-score provides an average of precision and recall.

5.2 Experimental Results and Discussion

In this section, we present and discuss the results of our evaluation (cf. Table 2 and 3). We recall edge type refers to the union of ordinary and inheritance edge types (cf. Definition 3.2).

5.2.1 Quality of the Schema Inference. 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 (cf. Table 4) demonstrate the overall good quality of the types inferred with our label-oriented approach. We could not compute these metrics for the Covid19 dataset due to the lack of a ground truth on this schema.

Dataset	Nodes	Edges	Node Labels	Edge Labels	Unlabel. 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 1: Characteristics of the datasets used in the study.

	Baseline		label-oriented-variant				
Dataset	Node Types	Edge Types	Node Types	Edges Types	Inheritance Edges Types	Max Node Hierarchies Depth	Overlapping Types
mb6	10	64	5	10	1	1	No
fib25	10	64	5	10	1	1	No
covid19	60	75	77	159	43	1	Yes
LDBC	7	21	7	21	0	0	No

Table 2: Inferred types with the label-oriented variant.

Dataset	Node Types	Edge Types	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 3: Inferred types with the property-oriented variant.

	Label-oriented						Property-oriented					
	Precision		Recall		F1		Precision		Recall		F1	
Dataset	N	E	N	E	N	E	N	E	N	E	N	E
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

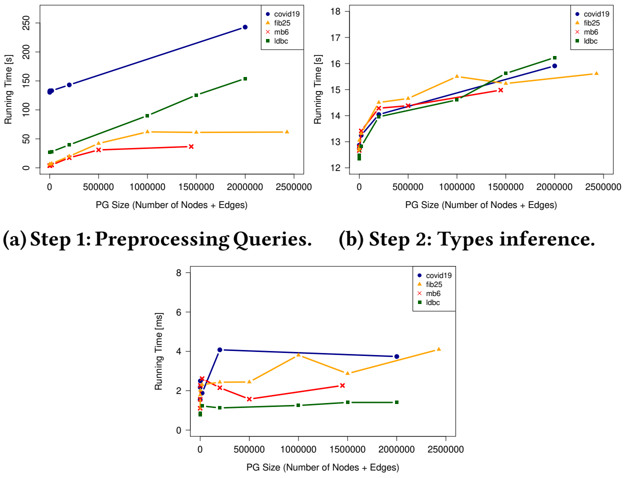
Table 4: Precision, recall and F1 of the inferred types (N is for node types, E is for edge types).

Baseline Comparison. We first compare the schemas inferred by our label-oriented approach with a baseline, the schemas returned by the Neo4j `call db.schema` query (cf. Table 2). The latter outputs many spurious types as it only targets single-labeled node types, even in the presence of multi-labeled node instances. Moreover, no property types or cardinality constraints can be captured, as opposed to our proposed method. As a result, the baseline schema is not accurate and error-prone.

Label-Oriented Approach. All three metrics—notably the precision, with a 0.80 to 1.00 range—are reasonably high. They are identical for both Neuprint datasets (mb6 and fib25), as they share both the ground-truth and inferred types. Only one node type and its corresponding inheritance edge type absent in the ground-truth schema have been mistakenly 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. Hence, our algorithm incorrectly aggregated them into distinct node types. This highlights the sensitivity of our method to noisy labels. A statistical approach to the type inference, such as clustering methods [9], would allow to group together nodes that share similar, but not identical, label sets. Combining it with graph embedding [15], which maps the input graph to a low-dimension space while preserving the inherent characteristics of the graph to the best possible extent, like in [11], 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 but 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 property-oriented variant.

Property-Oriented Approach. The low precision and F1 scores obtained with the property-oriented approach may stem from the inference of numerous spurious types—in addition to the correct ones. For instance, in the mushroombody dataset (mb6)



(a) Step 1: Preprocessing Queries.

(b) Step 2: Types inference.

(c) Step 3: Node hierarchy inference.

Figure 4: Average running times for various graphs.

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 (cf. Example 4.4.1). 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, the node types `Place` and `Person` both hold a property with the key `name`.

Nonetheless, in the Neuprint graphs, the number of TPs and FPs 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 label-oriented variant. They 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.

Label-Oriented vs Property-Oriented Approaches. To summarize, the label-oriented variant outputs schemas with a very good precision. However, it is missing node types that can only be inferred through property-related information. This is partially overcome by the property-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 label-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 property-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.

5.2.2 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 label-oriented implementation. Those from our property-oriented implementation are of the same order of magnitude. The first step (cf. Fig. 4a), 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. 4a to improve legibility). 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. 4b displays the sublinear behavior of the running times of Step 2, which is as expected with regards to [2]. Moreover, our parsing function has an average running time smaller than 2ms, which is satisfactory. On average, Step 3 (cf. Fig. 4c) runs in less than 5ms and is constant when the input PG size increases, which comforts our complexity analysis carried out in Section 4.3. Additionally, Step 2 and 3 are not sensitive to the heterogeneity in the complexity of data types and structures displayed in the different datasets.

6 Conclusion and Future Work

We have presented a novel end-to-end schema inference method for PGs that handles complex and nested property values, multi-labeled nodes, node hierarchies overlapping node types, edge cardinality constraints and optionality of properties.

We have proposed and empirically evaluated two variants that both scale well. The *label-oriented* variant provides an inferred schema of good quality. One of its main shortcomings is the loss of property co-occurrence information that could lead to additional supertype identification. This is resolved by our *property-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 non-trivial way to combine the outputs of these two variants to exclusively retain the wanted node types.

Our schema inference method remains sensitive to variation in the labels and property keys, be it due to inconsistent or multilingual naming. To overcome this, it would be interesting to consider a clustering step on the nodes and edges of the input instances, possibly combined with graph embeddings taking into account semantic information to simplify graph representations.

References

[1] Renzo Angles. 2018. The Property Graph Database Model. *Proceedings of the 12th Alberto Mendelzon International Workshop on Foundations of Data Management* (2018).

[2] Amine Baazizi, Dario Colazzo, Giorgio Ghelli, and Carlo Sartiani. 2019. Parametric schema inference for massive JSON datasets. *The VLDB Journal* 28, 4 (2019).

[3] Peter W. Battaglia and et al. 2018. Relational inductive biases, deep learning, and graph networks. *ArXiv* abs/1806.01261 (2018).

[4] Angela Bonifati, George Fletcher, Hannes Voigt, and Nikolay Yakovets. 2018. *Querying Graphs*. Vol. 10. Morgan & Claypool Publishers. 1–184 pages.

[5] Angela Bonifati, Peter Furniss, Alastair Green, Russ Harmer, Eugenia Oshurko, and Hannes Voigt. 2019. Schema Validation and Evolution for Graph Databases. In *Conceptual Modeling*. Springer International Publishing, Cham, 448–456.

[6] Redouane Bouhamoum, Kenza Kellou-Menouer, Zoubida Kedad, and Stéphane Lopes. 2018. Scaling Up Schema Discovery for RDF Datasets. In *2018 IEEE 34th International Conference on Data Engineering Workshops (ICDEW)*. 84–89.

[7] Olaf Hartig and Jan Hidders. 2019. Defining schemas for property graphs by using the GraphQL schema definition language. In *Proceedings of GRADES/NDA Workshop*.

[8] LDBC Social Network Benchmark task force. 2019. *The LDBC Social Network Benchmark (version 0.3.2)*. Technical Report.

[9] Artem Lutov, Soheil Roshankish, Mourad Khayati, and Philippe Cudré-Mauroux. 2018. StaTIX - Statistical Type Inference on Linked Data. In *Proceedings of Big Data*.

[10] Silvio Normey Gómez, Lorena Etcheverry, Adriana Marotta, Silvio Normey, and Mariano P Consens. 2018. Findings from Two Decades of Research on Schema Discovery using a Systematic Literature Review. In *AMW*.

[11] Benedek Rozemberczki, Ryan Davies, Rik Sarkar, and Charles Sutton. 2019. GEMSEC: Graph Embedding with Self Clustering. *ASONAM* (2019), 65–72.

[12] Siddhartha Sahu, Amine Mhedhbi, Semih Salihoglu, Jimmy Lin, and M. Tamer Özsu. 2020. The ubiquity of large graphs and surprising challenges of graph processing: extended survey. *VLDB J.* 29, 2-3 (2020), 595–618.

[13] Shin-ya Takemura and et al. 2015. Synaptic circuits and their variations within different columns in the visual system of Drosophila. *Proceedings of the National Academy of Sciences* 112, 44 (Nov 2015), 13711–13716.

[14] Shin-ya Takemura and et al. 2017. A connectome of a learning and memory center in the adult Drosophila brain. *eLife* 6 (Jul 2017).

[15] Q. Wang, Z. Mao, B. Wang, and L. Guo. 2017. Knowledge Graph Embedding: A Survey of Approaches and Applications. *IEEE TKDE* 29, 12 (2017), 2724–2743.

CONTENTS

Schema Inference for Property Graphs

Hanâ Lbath
Lyon 1 University, ENS Lyon
France
hana.lbath@ens-lyon.fr

Angela Bonifati
Lyon 1 University
France
angela.bonifati@univ-lyon1.fr

Russ Harmer
CNRS, ENS Lyon
France
russ.harmer@ens-lyon.fr

Abstract

Graphs are pervasive in many applications in which interconnected data are used to represent, explore and predict digital and real-world phenomena. Oftentimes, graph data comes without a predefined structure and in a constraint-less fashion, thus leading to inconsistency and poor quality. 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 obtained during the first step. The third step analyzes subtypes and supertypes to infer node hierarchies. We present our schema inference pipeline under two variants, namely a label- and a property-oriented variant. Finally, we experimentally evaluate and compare its scalability and accuracy on several real-life datasets. To the best of our knowledge, our work is the first to address schema inference for property graphs.

1 Introduction

Over the past decade, graph-based knowledge representation has been becoming increasingly popular, be it with the advent of graph databases [4] as an alternative to relational databases, or to model complex systems, from social and fraud detection networks to smart city grids or neuronal networks. Graphs are applicable to all settings in which interconnected data are used to represent, explore and predict digital and real phenomena. Understanding the connections in the data is fundamental for companies to carry out analytical and machine learning tasks.

Oftentimes, graph data comes without a structure and in a constraint-less fashion, thus leading to inconsistency and poor quality. According to a recent survey [12], the most recurrent task for users of real-world graphs is data integration. As a matter of fact, graphs naturally lend themselves to information reconciliation and integration. However, the integration of large-scale graphs, e.g. in knowledge bases such as DBPedia, Wikipedia or the more recent CovidGraph, might turn out to be incorrect and error-prone if not guided by means of schema constraints. These constraints are also important pillars for query optimization and metadata management, the latter being unexplored topics in graph databases. There are several models for representing graphs. Among the most popular is the property graph (PG) data model, which is a multigraph with both labeled nodes and edges, along with property value pairs associated to both. It has gained adoption with systems such as ArangoDB, HANA Graph, Neo4j, Oracle PGX, TigerGraph, Titan, etc.

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 has to be stabilized as in many data management settings. 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 based on types as 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 non-trivial and challenging task. Existing PG schema inference methods in available graph databases (such as Neo4j) are simplistic in that they can only output basic edge types and node types and do not take into account the complexity of the PG data model. In particular, they cannot handle complex data types, overlapping node types or node hierarchies. A principled approach to PG schema inference is currently missing and highly desirable given the interest in an ongoing ISO SC32/ WG3 standardization process of PG schemas, involving people from academia and industry in the LDBC community[1].

In this work, we address this problem and present a novel end-to-end schema inference method covering the entire spectrum of features of the PG data model. Our method tackles complex and nested property values, multi-labeled and unlabeled nodes, node hierarchies and overlapping node types as well as edge cardinality constraints and optionality of properties. We also introduce two variants of our method, a label-oriented and a property-oriented one and investigate their pros and cons.

To enable scalability, our method leverages a MapReduce approach [2] developed for schema inference from JSON datasets, which both aggregates types and identifies data types. However, the JSON data model and the PG data model are significantly different. While the former can be seen as an edge-labeled tree-structured data model, the latter is more expressive as it is a multi-graph with novel schema components such as subtyping and edge cardinality constraints. Because of that, schema inference for property graph is a challenging and non-trivial problem that we tackle in this paper for the first time.

The schema inference pipeline we designed[2] can be divided into three main steps. First, we employ Cypher queries to extract and serialize the nodes and edges of the input PG, in addition to gathering information needed to infer edge cardinality constraints. Cypher[3] is an open-source graph query language developed by Neo4j, inspired from SQL and adopted by several graph database vendors. 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

[1] More information can be found at: https://www.gqlstandards.org
[2] The source code is available at: https://gitlab.com/Hgit/pgsinference
[3] http://www.opencypher.org

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 datasets.

2 Related Work

Several vendors propose graph databases supporting the PG data model, such as Neo4j, Oracle PGX, TigerGraph, RedisGraph and GraphQL. Neo4j offers a limited possibility to view a schema of the database via a Cypher query that outputs a single-labeled directed graph displaying which types of nodes can be connected together and through which types of edges. However, there is a lack of support of the PG data model in its full potential [4]. In particular, 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, which is a crucial capability of the PG data model. Furthermore, properties, edge cardinality constraints and node type hierarchies are disregarded. Nonetheless, other Cypher queries output 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 [7] 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 [10]. Typical examples are clustering techniques, like [9] and [6], which infer *types* in RDF datasets. Both are based on the assumption that the more *properties* two *entities* share, the likelier they belong to the same *type*. To this end, they group entities according to a similarity metric. They also both handle hierarchical and overlapping types. In [6], a density-based clustering method, DBSCAN, is adopted. In [9] a faster and more accurate clustering method, called StaTIX, is proposed. It uses the cosine similarity metric and is based on community detection. However, none of these techniques are suitable for PGs.

In [2], 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 [2] does not tackle explicitly the problem of overlapping types. Due to the remarkable differences between the JSON data model and the PG data model, their method is not directly applicable to PGs.

In summary, none of the above approaches fully satisfy our criteria for PG schema inference: inference of types, basic and complex data types, overlapping types and node hierarchies, and, to the best of our knowledge, ours is the first work presenting a schema inference method specifically tailored to PGs.

3 Preliminaries

In this section, we recall the definition of property graphs [1, 4, 5] and extend a PG schema definition to best fit the schema inference process presented in this paper.

Let $\mathcal{O}$ be a set of *objects*, $\mathcal{L}$ be a finite set of *labels*, $\mathcal{K}$ be a set of *property keys*, and $\mathcal{N}$ be a set of *values*. We assume these sets to be pairwise disjoint.

Definition 3.1. A *property graph* is a structure $(V, E, \eta, \lambda, \nu)$ where

- $V \subseteq \mathcal{O}$ is a finite set of objects, called vertices;
- $E \subseteq \mathcal{O}$ is a finite set of objects, called edges;
- $\eta : E \to V \times V$ is a function assigning to each edge an ordered pair of vertices;
- $\lambda : V \cup E \to \mathcal{P}(\mathcal{L})$ is a function assigning to each object a finite set of labels (i.e., $\mathcal{P}(S)$ denotes the set of finite subsets of set S); and,
- $\nu : (V \cup E) \times \mathcal{K} \to \mathcal{N}$ is a partial function assigning values for properties to objects;

such that $V \cap E = \emptyset$ and the domain of ν is finite.

In [5], a Data Definition Language (DDL) for PGs is proposed where the obtained schema is a PG itself. We expand it to introduce *edge cardinality*, *subtypes* and *supertypes* and the concept of *inheritance edge type*. All these are key concepts to make our inferred PG schemas as expressive and accurate as possible.

Definition 3.2. (**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, $\mathcal{NT}$ a set of *node types*, $\mathcal{ET}$ a set of *edge types*.

- **Property type**: A property type is a pair (k, t), where $k \in \mathcal{K}$ is the property key and $t \in \mathcal{T}$ is its data type.
- **Element type**: An element type $b \in \mathcal{BT}$ is a quadruple (l, P, M, E), where $l \in \mathcal{L}$ is a label, P is a set of property types, $M \subseteq P$ is a subset of mandatory property types and $E \subseteq \mathcal{BT}$ is the set of element types that b extends.
- **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.
- **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.
- **Ordinary Edge Type**: An ordinary edge type is a quadruple (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**.
- **Edge Type**: the disjoint union of ordinary edge and inheritance edge types.

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

Definition 3.3. (**Overlapping Type**) An overlapping type is an element type which is a subtype of two or more supertypes.

4 Inferring Property Graph Schemas

In this section we present our method to infer a PG schema. We assume all nodes and edges in the PG are labeled. Nodes are of the same type if and only if they have the same set of labels, while edges are of the same type if and only if they share their set of source nodes, target nodes and edge labels. These assumptions may be too strong in some cases. We will present in Section 4.4 an alternative that deals with some of its shortcomings.

Our PG schema inference pipeline can be divided into three main steps (cf. Fig. 1).

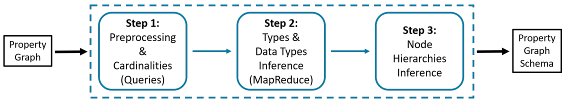

Property Graph
Step 1: Preprocessing & Cardinalities (Queries)
Step 2: Types & Data Types Inference (MapReduce)
Step 3: Node Hierarchies Inference
Property Graph Schema

Figure 1: Schema inference main steps.

4.1 Step 1: Preprocessing

To begin with, the input PG 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.

4.1.1 Preprocessing Queries. First, nodes are matched according to their labels via Cypher queries. Similarly, edges are matched according to their source, target node and edge labels. Node and edge types such that none of their instances have properties are stored separately. As there is no need to infer their property data types, the MapReduce step can be skipped in these cases and they can be directly processed by the third step.

4.1.2 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.

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 *optional* for the source node type and *mandatory* for the target node type).

4.1.3 Serialization to JSON. The Cypher queries output node and edge `neo4j` objects where property values are sometimes 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 dictionaries that are stored in `jsonline` files in such a way that correct type inference by the MapReduce method is guaranteed (cf. Section 4.2). 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. Similarly, the edges file contains a single dictionary where each key-value pair represents 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.

4.1.4 Example. Let us set a PG instance G of a social network of patients and doctors who can create and like posts and comments as well as reply to them. This is partially inspired by the LDBC Social Network Benchmark database [8]. Listings 1 and 2 are dictionaries encoding a node and edge instance.

```
{'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 containing edge cardinality information but no properties (they will be added in Step 2).

4.2 Step 2: Types and Data Types Inference (MapReduce)

In this step, we aggregate nodes and edges by type and infer the property data types by relying on MapReduce [2]. It can be summarized in two steps : i) a **Map** phase where all property value data types are inferred (cf. Listing 3) and ii) a **Reduce** phase where types are fused according to an equivalence relation. Here, we use the *kind-equivalence* relation described in [2] (cf. Listing 4). It fuses recursively types of the same *kind*, i.e. records with records, arrays with arrays and basic types (`String`, `Number` and `Boolean`) with basic 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, 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 4. The fusion function is recursively called so as to handle nested values.

The output of the algorithm, which is a JSON record, is then parsed and stored in a human-readable dictionary where question marks are affixed to optional properties' data types. A property "`meta_mandatory` : False" 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. Thus, the output of this step is a preliminary PG schema of the input PG, but it is still missing subtyping information.

4.2.1 Example. Listing 3 and 4 illustrate the fusion of two {`Person`, `Patient`} nodes 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: Fusion of the two JSON record types on the left-hand side using the *kind-equivalence*.

4.3 Step 3: Inference of Node Hierarchies

The final step is to infer node type hierarchies so as to obtain a schema satisfying Definition 3.2. Inferring edge hierarchies is unnecessary in Neo4j graphs, since edges can only be associated with a single label. Nevertheless, we expect our hierarchy inference technique to straightforwardly extend to edge hierarchies.

Algorithm 1: Node Hierarchy Inference (Label-Oriented Variant)

```
   input : schema nodes and edges dictionaries from Step 2
   output: updated schema nodes and edges dictionaries and schema file
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 ← 0 to length(nodeLabels) − 1 do
8      nset0 = nodeLabels[i]
9      for j ← 0 to length(nodeLabels[i :]) − 1 do
10                                 ▷ Two given node types are compared only once
11         nset1 = nodeLabels[j]
12         if nset0 ≠ nset1 then
13             if nset0 ⊂ nset1 then
14                 add nset1::SubtypeOf::nset0 edge
15             else if nset1 ⊂ nset0 then
16                 add nset0::SubtypeOf::nset1 edge
```

First, node **supertypes** corresponding to subsets of labels present in two or more node type 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. 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 thus 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 quadratic in the number of node types inferred in the previous steps. However, since the number of node types is typically smaller than the size of the PG, this complexity remains bearable in practice, as also shown by our evaluation.

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

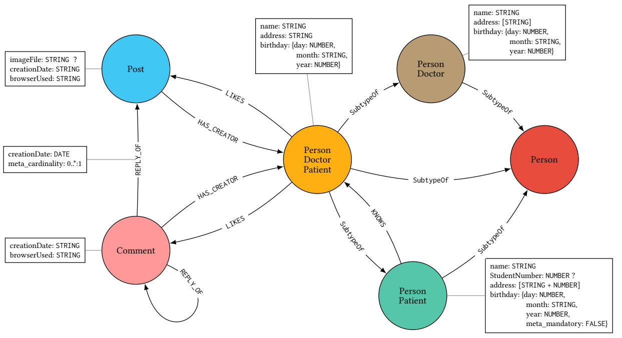


Figure 2: An excerpt of the PG schema inferred from the PG G using our label-oriented variant.

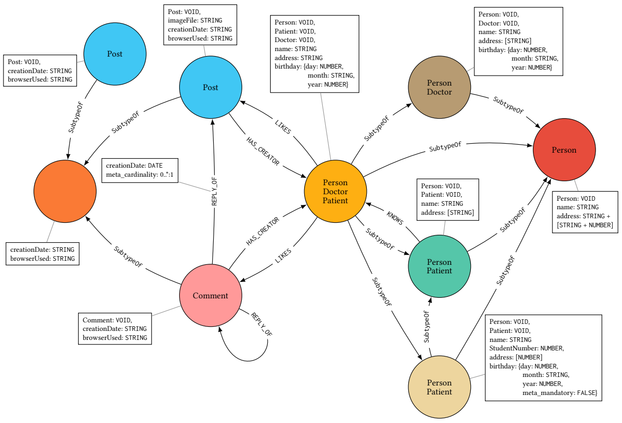


Figure 3: An excerpt of the PG schema inferred from the PG G using our property-oriented variant.

4.4 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. Example 4.1.4) 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 and Fig. 2). 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 thus identify node types and hierarchies. To merge nodes, we hence use $\mathcal{L}$*-driven reduction* [2], 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 label-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.

4.4.1 Example. Fig. 2 and 3 depict the schemas inferred from the PG *G* using both variants. As discussed above, the first approach overlooks the information provided by the properties, resulting in missing node types (e.g., a supertype of the `Post` and `Comment` nodes could not be inferred). This is resolved with the property-oriented variant where an unlabeled supertype could be inferred (the unlabeled orange node in Fig. 3).

5 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. The MapReduce step is based on the implementation in [2], 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.

5.1 Datasets and Metrics

We have used several datasets in our experimental study (cf. Table 1). We evaluated our schema inference method on the **LDBC Social Network Benchmark** (**LDBC**) [8], a synthetic social network. It contains single-labeled nodes and comes equipped with a ground truth schema incorporating node hierarchies. We also used two **Neuprint** datasets, corresponding to neuronal networks of different parts of the fruit fly brain: i) the mushroombody (**mb6**) [14] and ii) the medulla (**fib25**) [13]. Accompanied by a ground truth schema, they contain multi-labeled nodes (as opposed to LDBC) and a large diversity of property value data types, such as JSON records or `neo4j` cartesian 3D points. We tested as well our pipeline on a **Covid-19 graph** (**covid19**) (https://covidgraph.org/). This graph is being assembled by the CovidGraph project, which is currently ongoing. 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. No ground truth is available for the schema of the latter graph.

To assess the quality of our schema inference, we have used the precision, recall and F1-score of the node types and edge types. We consider an inferred type that is (not) present in the ground truth schema as a True Positive (TP) (False Positive (FP), respectively). 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 are present in the ground truth, while the recall gives the proportion of ground truth types that were inferred. The F1-score provides an average of precision and recall.

5.2 Experimental Results and Discussion

In this section, we present and discuss the results of our evaluation (cf. Table 2 and 3). We recall edge type refers to the union of ordinary and inheritance edge types (cf. Definition 3.2).

5.2.1 Quality of the Schema Inference. 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 (cf. Table 4) demonstrate the overall good quality of the types inferred with our label-oriented approach. We could not compute these metrics for the Covid19 dataset due to the lack of a ground truth on this schema.

Dataset	Nodes	Edges	Node Labels	Edge Labels	Unlabel. 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 1: Characteristics of the datasets used in the study.

	Baseline		label-oriented-variant				
Dataset	**Node Types**	**Edge Types**	**Node Types**	**Edges Types**	**Inheritance Edges Types**	**Max Node Hierarchies Depth**	**Overlapping Types**
mb6	10	64	5	10	1	1	No
fib25	10	64	5	10	1	1	No
covid19	60	75	77	159	43	1	Yes
LDBC	7	21	7	21	0	0	No

Table 2: Inferred types with the label-oriented variant.

Dataset	Node Types	Edge Types	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 3: Inferred types with the property-oriented variant.

	Label-oriented						Property-oriented					
	Precision		Recall		F1		Precision		Recall		F1	
Dataset	N	E	N	E	N	E	N	E	N	E	N	E
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

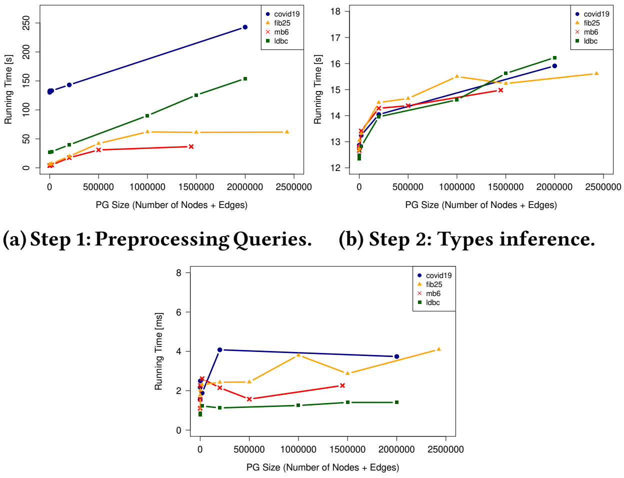
Table 4: Precision, recall and F1 of the inferred types (N is for node types, E is for edge types).

Baseline Comparison. We first compare the schemas inferred by our label-oriented approach with a baseline, the schemas returned by the Neo4j `call db.schema` query (cf. Table 2). The latter outputs many spurious types as it only targets single-labeled node types, even in the presence of multi-labeled node instances. Moreover, no property types or cardinality constraints can be captured, as opposed to our proposed method. As a result, the baseline schema is not accurate and error-prone.

Label-Oriented Approach. All three metrics—notably the precision, with a 0.80 to 1.00 range—are reasonably high. They are identical for both Neuprint datasets (mb6 and fib25), as they share both the ground-truth and inferred types. Only one node type and its corresponding inheritance edge type absent in the ground-truth schema have been mistakenly 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. Hence, our algorithm incorrectly aggregated them into distinct node types. This highlights the sensitivity of our method to noisy labels. A statistical approach to the type inference, such as clustering methods [9], would allow to group together nodes that share similar, but not identical, label sets. Combining it with graph embedding [15], which maps the input graph to a low-dimension space while preserving the inherent characteristics of the graph to the best possible extent, like in [11], 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 but 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 property-oriented variant.

Property-Oriented Approach. The low precision and F1 scores obtained with the property-oriented approach may stem from the inference of numerous spurious types—in addition to the correct ones. For instance, in the mushroombody dataset (mb6)



(a) Step 1: Preprocessing Queries.

(b) Step 2: Types inference.

(c) Step 3: Node hierarchy inference.

Figure 4: Average running times for various graphs.

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 (cf. Example 4.4.1). 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, the node types `Place` and `Person` both hold a property with the key `name`.

Nonetheless, in the Neuprint graphs, the number of TPs and FPs 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 label-oriented variant. They 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.

Label-Oriented vs Property-Oriented Approaches. To summarize, the label-oriented variant outputs schemas with a very good precision. However, it is missing node types that can only be inferred through property-related information. This is partially overcome by the property-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 label-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 property-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.

5.2.2 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 label-oriented implementation. Those from our property-oriented implementation are of the same order of magnitude. The first step (cf. Fig. 4a), 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. 4a to improve legibility). 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. 4b displays the sublinear behavior of the running times of Step 2, which is as expected with regards to [2]. Moreover, our parsing function has an average running time smaller than 2ms, which is satisfactory. On average, Step 3 (cf. Fig. 4c) runs in less than 5ms and is constant when the input PG size increases, which comforts our complexity analysis carried out in Section 4.3. Additionally, Step 2 and 3 are not sensitive to the heterogeneity in the complexity of data types and structures displayed in the different datasets.

6 Conclusion and Future Work

We have presented a novel end-to-end schema inference method for PGs that handles complex and nested property values, multi-labeled nodes, node hierarchies overlapping node types, edge cardinality constraints and optionality of properties.

We have proposed and empirically evaluated two variants that both scale well. The *label-oriented* variant provides an inferred schema of good quality. One of its main shortcomings is the loss of property co-occurrence information that could lead to additional supertype identification. This is resolved by our *property-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 non-trivial way to combine the outputs of these two variants to exclusively retain the wanted node types.

Our schema inference method remains sensitive to variation in the labels and property keys, be it due to inconsistent or multilingual naming. To overcome this, it would be interesting to consider a clustering step on the nodes and edges of the input instances, possibly combined with graph embeddings taking into account semantic information to simplify graph representations.

References

[1] Renzo Angles. 2018. The Property Graph Database Model. *Proceedings of the 12th Alberto Mendelzon International Workshop on Foundations of Data Management* (2018).

[2] Amine Baazizi, Dario Colazzo, Giorgio Ghelli, and Carlo Sartiani. 2019. Parametric schema inference for massive JSON datasets. *The VLDB Journal* 28, 4 (2019).

[3] Peter W. Battaglia and et al. 2018. Relational inductive biases, deep learning, and graph networks. *ArXiv* abs/1806.01261 (2018).

[4] Angela Bonifati, George Fletcher, Hannes Voigt, and Nikolay Yakovets. 2018. *Querying Graphs*. Vol. 10. Morgan & Claypool Publishers. 1–184 pages.

[5] Angela Bonifati, Peter Furniss, Alastair Green, Russ Harmer, Eugenia Oshurko, and Hannes Voigt. 2019. Schema Validation and Evolution for Graph Databases. In *Conceptual Modeling*. Springer International Publishing, Cham, 448–456.

[6] Redouane Bouhamoum, Kenza Kellou-Menouer, Zoubida Kedad, and Stéphane Lopes. 2018. Scaling Up Schema Discovery for RDF Datasets. In *2018 IEEE 34th International Conference on Data Engineering Workshops (ICDEW)*. 84–89.

[7] Olaf Hartig and Jan Hidders. 2019. Defining schemas for property graphs by using the GraphQL schema definition language. In *Proceedings of GRADES/NDA Workshop*.

[8] LDBC Social Network Benchmark task force. 2019. *The LDBC Social Network Benchmark (version 0.3.2)*. Technical Report.

[9] Artem Lutov, Soheil Roshankish, Mourad Khayati, and Philippe Cudré-Mauroux. 2018. StaTIX - Statistical Type Inference on Linked Data. In *Proceedings of Big Data*.

[10] Silvio Normey Gómez, Lorena Etcheverry, Adriana Marotta, Silvio Normey, and Mariano P Consens. 2018. Findings from Two Decades of Research on Schema Discovery using a Systematic Literature Review. In *AMW*.

[11] Benedek Rozemberczki, Ryan Davies, Rik Sarkar, and Charles Sutton. 2019. GEMSEC: Graph Embedding with Self Clustering. *ASONAM* (2019), 65–72.

[12] Siddhartha Sahu, Amine Mhedhbi, Semih Salihoglu, Jimmy Lin, and M. Tamer Özsu. 2020. The ubiquity of large graphs and surprising challenges of graph processing: extended survey. *VLDB J.* 29, 2-3 (2020), 595–618.

[13] Shin-ya Takemura and et al. 2015. Synaptic circuits and their variations within different columns in the visual system of Drosophila. *Proceedings of the National Academy of Sciences* 112, 44 (Nov 2015), 13711–13716.

[14] Shin-ya Takemura and et al. 2017. A connectome of a learning and memory center in the adult Drosophila brain. *eLife* 6 (Jul 2017).

[15] Q. Wang, Z. Mao, B. Wang, and L. Guo. 2017. Knowledge Graph Embedding: A Survey of Approaches and Applications. *IEEE TKDE* 29, 12 (2017), 2724–2743.