Finding the PG schema of any (semi)structured dataset: a tale of graphs and abstraction
Nelly Barret, Tudor Enache, Ioana Manolescu, Madhulika Mohanty

To cite this version:
Nelly Barret, Tudor Enache, Ioana Manolescu, Madhulika Mohanty. Finding the PG schema of any (semi)structured dataset: a tale of graphs and abstraction. SEAGraph Workshop 2024 - 3rd Workshop on Search, Exploration, and Analysis in Heterogeneous Datastores - 40th IEEE International Conference on Data Engineering (ICDE 2024), May 2024, Utrecht, Netherlands. hal-04591933

HAL Id: hal-04591933
https://inria.hal.science/hal-04591933
Submitted on 29 May 2024

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution - NonCommercial - ShareAlike 4.0 International License
Finding the PG schema of any (semi)structured dataset: a tale of graphs and abstraction

Nelly Barret*, Tudor Enache†, Ioana Manolescu* and Madhulika Mohanty*
* Inria & Institut Polytechnique de Paris, France
{firstname.lastname}@inria.fr
† Ecole Polytechnique, France
tudor.enache@polytechnique.edu

Abstract—Property Graphs (PGs) are an attractive data model both for business users, and for developers of data management tools. They combine the internal structure helpful in relational databases, where each record has a clearly identified set of attributes, with the flexible structure and support for heterogeneity, common in graph databases.

Several useful and/or interesting datasets are available in non-PG data models. These include legacy databases, created before the advent of the PG standards, as well as well-known benchmarks based on real and synthetic data, Open Data published in other formats such as XML, JSON or RDF, etc. Converting such datasets to Property Graphs would enable their exploitation under the PG model.

In this work-in-progress paper, we describe an approach to derive, from any (semi)-structured dataset, a PG schema consisting of node types, edge types, and a graph type. Our approach builds on (i) ConnectionLens, a tool for converting (semi)-structured datasets into simple data graphs, and (ii) Abstra, which, in a ConnectionLens graph, identifies a set of entities and relationships. This work is the first step towards a universal data migration tool from (semi)-structured data, to PGs.

I. MOTIVATION AND OUTLINE

There is an unprecedented creation of data pertaining to various contexts, like health, finance or social networks. Even though the W3C recommends sharing the data as RDF graphs, practitioners still use other models, e.g., relational, XML, JSON, etc. While semi-structured data models are very flexible and enable describing data with varied structure, datasets shared under these forms may be hard to understand for new users, especially if the data is shared with insufficient or no documentation. To solve this problem, there have been several efforts to generate (infer) a schema from the data itself. Existing schema generation approaches are each designed for a given data model, e.g., for JSON [4], [3], [16], [27], XML [10], and RDF [13]. In a similar but different vein, ABSTRA [5] generates, from a (semi-)structured dataset of one among multiple models, a dataset abstraction, akin to the traditional Entity-Relationship diagrams [23], but also allowing deeply nested entities. An abstraction is data model-independent; it is not a grammar, but a diagram, giving users a first look at the dataset.

More recently, Property Graphs (PGs, in short) are adopted in various application domains, e.g., the International Consortium of Investigative Journalism (ICIJ) built the Offshore leaks PG database [30], which has been used to detect tax evasion. Numerous industrial PG databases exist, e.g., from Neo4J [19] and Oracle [20]. Property graphs consist of labeled nodes, possibly connected by labeled edges; both nodes and edges may have attributes in the form of key/value pairs. This follows the “record-style” of existing relational databases (an object has a label and some attributes), while accounting for semi-structured data heterogeneity and complexity (not all records have the exact same set of attributes). Also, edges may have their own attributes, something that is not possible in relational model. The popularity of PGs has led to efforts to standardize the data model, the query language, and to generate a schema from a given graph [7], [15]. A recent, comprehensive proposal for PG schemas (in the classical database sense: schema defined independently of a dataset, introducing types that the dataset may or may not validate) is [2]. In order to be able to exploit these benefits, many prior works have targeted the problem of automatically converting a relational dataset into a graph [25], [28].

In this work-in-progress paper, given any semi-structured dataset, we aim to derive a PG schema for it. We achieve this in the following steps:

(i) Create a simple data graph out of the semi-structured dataset using CONNECTIONLENS [1] (Sec. II-A);
(ii) Abstract this data graph to detect the entities and relationships it describes, using ABSTRA [5] (Sec. II-B);
(iii) Derive a PG type, as described in [2], from these entities and relationships (Sec. III). This is the main contribution of this paper.

We also evaluate the quality and soundness of the generated PG schemas on several (semi)-structured datasets (Sec. IV). Finally, we conclude and provide future extensions that could be built on top of our PG schema generator for (semi)-structured data (Sec. V).

II. BACKGROUND

A. From (semi-)structured data to a data graph

CONNECTIONLENS [1] (CL, in short) is a system which, starting from any (set of) structured, semi-structured or unstructured datasets, converts it to a simple data graph, with atomic nodes, and labeled nodes and edges. In this graph, nodes have a unique ID and a label (possibly empty), edges have a unique ID, a source node, a target node, and a label (also possibly empty). Thus, CONNECTIONLENS constructs
$G = (N, E, \lambda)$ where $N$ is the node set, $E \rightarrow N \times N$ is the edge set, and $\lambda$ is a function labeling nodes and edges, possibly with the empty label $\epsilon$. CL constructs a simple data graph as follows. XML documents translate into trees, where each element node, respectively element or attribute value leads to a node in $G$. Edges are modeling the parent-child relationships. An edge connecting an element node to an attribute value is labelled with that attribute name; other edges are labelled $\epsilon$. When an XSD [31] accompanies the data, ID-IDREF connections lead to an edge between the IDREF node to the ID node, thus the resulting graph $G$ is no longer a tree. JSON documents also lead to trees, where each map, array and (map or array) value is modelled as a node. A map node is connected to each of its attribute values by an edge labelled with the attribute name, while an array node is connected to its value using an $\epsilon$-labelled edge. RDF graphs are easily converted to simple graphs: each triple $\langle s \mid p \mid o \rangle$ leads to a $p$-labelled edge connecting a node labelled $s$ to a node labelled $o$. For CSV tables, a node is created for each line (tuple), respectively. If a header was present, edges connecting lines to their value are labelled with the corresponding header name, otherwise the edge is $\epsilon$-labelled. We call value nodes the data nodes created out of XML (element or attribute) values, JSON (attribute or array) value, RDF literals and CSV values. Those values are constants. Others nodes, e.g., XML elements, JSON map or array, etc. are structural nodes as they organize the data.

B. From a data graph to an abstraction [5]

To better understand a dataset, and towards identifying entities and their possible relationships, ABSTRA [5] builds an abstraction thereof, as follows. First, it summarizes the simple data graph $G$ based on an equivalence relation among the nodes in the graph. The resulting summary is a (much smaller) graph $\mathcal{G}$, each of whose nodes corresponds to a set (or collection) of nodes from $G$; for each edge in $G$, $\mathcal{G}$ has an edge between the respective two $\mathcal{G}$ nodes. We call $\mathcal{G}$ a collection graph, and its elements collection nodes, respectively, collection edges. Each data model is summarized with the equivalence relation best suited to it. Thus, it considers equivalent: XML nodes having the same label, JSON and CSV nodes on the same path from the root. For RDF nodes, summarization relies on a flexible, type-and-structure-based equivalence relation introduced in prior work [12].

Next, ABSTRA selects a set of collection nodes $E \subseteq \mathcal{G}$ to be promoted as (main) “entities”; the remaining $\mathcal{G}$ nodes will either be considered attributes of one or several entities in $E$, or found to describe relationships between these entities. Users can limit the size of $E$, in which case ABSTRA will reflect only the entities containing “most” data nodes. Without this limit, all the dataset is reflected in the returned entities, whose number depends on the dataset’s structural complexity. For instance, out of an XMark [24] XML document of 5M nodes describing an auction website, given a limit of 5 entities, ABSTRA identifies: open_auction, closed_auction, item, person and category records (the five boxes in Fig. [1]).

A boundary is then computed for each such main entity: a set of $\mathcal{G}$ nodes considered to be part of (attributes belonging to) the main entity, and the edges connecting these nodes to each other, and to the main entity. While in classical E-R design [23] all entity attributes have atomic values, attributes of these entities can be nested. For instance, the boundary of the person entity includes: name, emailaddress, id, homepage, phone, creditcard, and address; the latter has the nested attributes province, city, zipcode, country and street (not shown in Fig. [1]).

Third, to each main entity is assigned a semantic class from an ontology built based on open Knowledge Bases (KB) and other linguistic resources, leveraging the labels of the nodes in the entity (node collection) and/or the labels of their attributes. For instance, the item entity is classified as a Product, mainly because it is labelled item, it has quantity and shipping attributes. In the last step, a set of relationships $\mathcal{R}$ connecting the main entities is identified based on the $\mathcal{G}$ paths connecting the main entity nodes. For instance, an item has a category and an open_auction refers to an item.

C. PG Schema language [2]

We recall the recent PG schema [2] proposal on which we build our work. Let $\mathcal{L}$ be a set of (node and edge) labels; a PG schema graph type $T_{G}$ consists of a set of node types $T_{N}$ and a set of edge types $T_{E}$. A node type $T_{N}^{i}$ specifies a certain set of node labels $L^{i} \subseteq \mathcal{L}$ and set of attributes $A^{i} \subseteq A$. An edge type $T_{E}^{j}$ is characterized by a set of node labels $L^{j} \subseteq \mathcal{L}$, a set of edge attributes $A^{j} \subseteq A$, and a pair of node types for its source and destination nodes, $T_{N}^{s}$ and $T_{N}^{d}$; this is written $T_{E}^{j}=(T_{N}^{s}:L^{j}:A^{j})$. Any node/edge attribute is atomic, that is, its values can only be constants. An attribute may be declared as OPTIONAL. The labels and attributes for a certain node/edge type can be specified as OPEN to indicate that nodes/edges of this type can also have other labels (resp., attributes) not explicitly specified in the schema; by default, they are not open. The graph type $T_{G}$ can be specified either as STRICT (each nodes and edges must validate at least one of the corresponding specified types) or LOOSE (some nodes/edges may not comply to any type in the schema), the latter giving more flexibility.

III. Deriving a PG Schema for any Dataset

We now present our method for deriving, for any dataset (or set of datasets), a PG schema [2] starting from their abstraction, such that (i) the data conforms to the PG schema (it could be entirely converted into a PG graph valid w.r.t. the schema), and (ii) the PG schema is relatively “tight”, i.e., only datasets structurally similar to the input one would be valid w.r.t. the schema. With this goal in mind, our target PG Schema is not OPEN.

Beyond the entities $E$ and relationships $\mathcal{R}$, our algorithm (Algo. [1]) takes a parameter $\phi \in \{\text{FLAT}, \text{CUT}\}$, specifying how to map possible nested ABSTRA attributes into PG schema node/edge attributes. Intuitively, we can either (i) “wrap” all
the content of the nested attribute into a single atomic one. At the data level (not discussed in this paper where we only synthesize a PG schema), this corresponds to a traversal (and serialization) of the nested ABSTRA entity attribute value, into a single field, that will be the value of the PG schema attribute (FLAT); or (ii) “cut (separate)” the nodes in the nested entity attribute, into as many standalone PG node types as needed (CUT).

First, Algo. 1 identifies a set of PG node types. For each entity \( e \) in \( E \), we compute a node type \( T^e_N \) to which we associate a set of labels \( L^e \) and a set of attributes \( A^e \) (Sec. II-C). The node type and node label(s) are already provided by ABSTRA: they correspond to the entity name (the natural common collection name), respectively its semantic class (Sec. II-B). In our case, \(|L^e|=1\) because ABSTRA assigns only one semantic class to an entity. For instance, in Fig. 1 the light blue entity leads to \( T^e_N=\text{item} \) and \( L^e=\text{Product} \). Next, to build the attribute set \( A^e \), we iterate over each attribute \( a \in e \), and proceed as follows (Lines 7-13): (i) if \( a \) has atomic values, we simply add it to \( A^e \), declaring it of type string; (ii) otherwise, we decide based on \( \phi \). If \( \phi=\text{FLAT} \), the attribute, with all its descendants that are still in the boundary of the main entity, are wrapped in an atomic value (Line 11). In Fig. 1 the item’s attribute description would lead to a JSON object containing text, ul and li (the description attributes; hidden in Fig. 1). Otherwise (\( \phi=\text{CUT} \)), the attribute is unfolded: for each of its child attributes, a (new) node type is generated, as well as the corresponding “parent-child” edge types are created (Line 13). Using CUT, the item description attribute would lead to a new PG node type, having attributes text, ul and li; a PG edge type would connect item to description. Further, \( a \) is marked as OPTIONAL when not all records have it (Line 15). For instance, only few items have a shipping element, thus it is marked as optional.

Second, we compute PG edge types. For each pair of ABSTRA entities \( e_i \) and \( e_j \) connected by a \( l \)-labeled relationship, we get the PG node types of the source and target entities \( (T^e_N \text{ and } T^e_N) \) and add to the PG schema an edge type \( T^e_E \) labeled with the ABSTRA relationship label \( l \). For instance, in Fig. 1 the entity person is connected to the entity open_auction with a relationship labeled watches.watchopen_auction. Thus, in the PG schema, the corresponding PG node types personType and open_auctionType are connected by a PG edge type Edge3Type, labeled Watches_watchOpen_auction (Fig. 2).

For what concerns the PG graph type, if the abstraction represents 100% of the data (recall Sec. II-B), we declare the schema to be STRICT. Otherwise (if abstraction left some data out because of a limit on the size of \(|E|\)), the resulting PG schema is LOOSE. This is because the unrepresented nodes, respectively, edges, will not comply with any of the corresponding types defined in the PG schema.

Fig. 2 shows part of the PG schema obtained from the XMark abstraction (Fig. 1) with \( \phi=\text{FLAT} \). The first node type personType comes from the abstraction itself (entity person), while the addressType comes from a nested attribute flattened as a String. An edge type is defined to connect the personType to the addressType. More edge types (personType to categoryType and personType to open_auctionType) have been declared, following ABSTRA relationships.

IV. EVALUATION

We implemented the PG schema generation algorithm in Python. Its starting point is an ABSTRA-computed abstraction, as well as the simple data graph, stored in a Postgres database.

A. Datasets

We tested our PG schema generation approach on several datasets (Tab. 1), of different data models. The Companies dataset (CSV) describes the 40 most influential French companies by their id, name and Wikipedia headline; Conferences (RDF) is about scientific publications (having a title and year) and their authors (identified by their first and last names and affiliation); the JSON Researchers dataset describes authors (id, first and last names, gender, age, status) and their top-5 publications as well as their 3 most frequent 3 co-authors. Finally, XML evaluation datasets comprise: an XMark dataset (Fig. 1); the HATVP dataset [14], a French public transparency dataset about elected officials’ wealth; PubMed one is a sample of bibliographic notices available in PubMed, a repository of scientific biomedical literature. Double arrows (\( \leftrightarrow \)) indicate datasets including entities with nested attributes, while real-life datasets are denoted by a \( \bullet \).
Algorithm 1: ABSTRA abstraction to PG schema

Input: N, E, R, φ ∈ {FLAT, CUT}
Output: PG schema TG
1 \(T_N \leftarrow \emptyset, T_E \leftarrow \emptyset;\)
2 for \(e \in E\) do
3 \(T^e_N \leftarrow e.name;\)
4 \(L^e \leftarrow e.semantic_class;\)
5 \(A^e \leftarrow \emptyset;\)
6 for attribute \(a \in e.boundary\) do
7 if \(a\) is not a nested attribute then
8 \(A \leftarrow A + a \text{ STRING};\)
9 else
10 if \(φ=\text{FLAT}\) then
11 \(A \leftarrow \text{wrap}(a);\)
12 else
13 \(A \leftarrow \text{unfold}(a);\)
14 if all nodes in the collection \(e\) do not have \(a\) then
15 \(A \leftarrow \text{OPTIONAL } A;\)
16 \(A^e \leftarrow A^e \cup \{A\};\)
17 \(T_N \leftarrow T_N \cup \{T^e_N : L^e A^e\};\)
18 \(z \leftarrow 1;\)
19 for \(e_i \rightarrow e_j \in R\) do
20 \(T_E \leftarrow T_E \cup \{T^e_N \cup (T^e_N : l) \rightarrow (T^e_N)\};\)
21 \(z \leftarrow z + 1;\)
22 if \(E\) and \(R\) represent all \(N\) and \(E\) resp. then
23 \(TG \leftarrow \text{STRIG}(T_N, T_E);\)
24 else
25 \(TG \leftarrow \text{LOOSE}(T_N, T_E);\)
26 return \(TG;\)

B. Metrics

We evaluated each generated PG schema on the following points:
(i) **Size**: How do they compare to abstractions in terms of size? (Sec. IV-C);
(ii) **Correctness**: Are the PG schemas syntactically correct? (Sec. IV-D); and
(iii) **Soundness**: Are they true to the initial abstraction? (Sec. IV-D).

We did not report on scalability as the time spent to generate the PG schemas was not significant (usually, less than a second in our experiments).

C. Data abstractions vs PG schemas size

To answer (i), Tab. 1 shows, for each dataset, the number of nodes and edges in the simple data graph \(|N|\) and \(|E|\) in Sec. II-A; the number of ABSTRA entities and relationships \((E\) and \(R\) in Sec. II-B); the number of nodes and edges in the PG schema with \(φ=\text{FLAT}\) \(|N^F|\) and \(|E^F|\), respectively for \(φ=\text{CUT}\) \(|N^C|\) and \(|E^C|\).

When a data abstraction contains only simple attributes, the resulting PG schema is of the same size, regardless of the value of \(φ\). This is because no attribute leads to new (additional) node types, as in the Companies and Conferences datasets.

In contrast, when the abstraction features entities with nested attributes: (i) When \(φ=\text{FLAT}\), the PG schema is of the same size as the data abstraction in terms of nodes \(|N^F|\) and edges \(|E^F|\). This is the case of all datasets with nested attributes (\(\mathbb{1}\)). (ii) When \(φ=\text{CUT}\), the PG schema is larger than the abstraction, both in terms of nodes and edges, because new PG node and edge types are created out of the nested attributes, as in the HATVP dataset where more than 200 of each have been created. This is because the dataset is a deep tree, where some attributes have up to 69 child attributes (themselves containing few attributes), all leading to new PG node types.

D. Correctness and soundness of the generated schemas

To answer (ii), we parsed our generated PG schemas using ANTLR [29] and verified that all of them are successively accepted by the grammar outlined in [2]. To answer (iii), 3 authors compared manually the abstraction E-R diagram and the generated PG schema, and answered the following questions:

(i) **Are all ABSTRA entities represented in the PG schema?**;
(ii) **Do attributes belong to the right entity?**;
(iii) **Are nested attributes faithfully represented in the PG schema?**; and
(iv) **Are relationships connecting the right entities with the right label?**.

The three authors have unanimously answered “Yes” to all the questions indicating that the generated PG schemas faithfully represent the initial data abstraction, including the nested elements.

V. CONCLUSION AND FUTURE WORK

We presented an approach to derive, from any (semi-)structured dataset, a PG schema following the syntax described in [2]. Among the closest related work, the W3C defined a language for expressing relational-to-RDF mappings (R2RML [22]). There have been prior works that recommend a relational schema for a semi-structured dataset, e.g. [11],[26], [6], [8]. However, to the best of our knowledge, our work is the first to aim at mapping data from a variety of formats, into PGs. To this end, we exploit simple data graphs built by CL [1], and data abstractions of [5]. Mapping heterogeneous datasets to a well-defined schema also facilitates information
This work is partially funded by DIM RFSI PHD 2020-01, AI Chair SourcesSay (ANR-20-CHIA-0015-01) and European Commission under Horizon Europe Programme (ELIAS-101120237).

REFERENCES


