From simple graphs to property graphs

Enache Tudor

École Polytechnique

Abstract. Property graphs are an increasingly popular format for representing data, consisting of labeled nodes connected by possibly labeled edges in such a way that edges are stored natively alongside the nodes in a flexible format. The implementation of property graphs has been accompanied by an effort of standardisation, giving rise to a set of competing proposals for a schema of property graphs. Abstraction is a technique of data summarisation which differs from schemas in that it is aimed at human users, while schemas are used by code that processes data. However, both techniques share the goal of compactly describing a dataset. Starting from the ABSTRA system [1] for abstracting databases and the PG-SCHEMA proposal [2], we detail the development of an algorithm for creating a schema starting from an abstraction. We test our algorithm on five datasets stored in diverse formats (RDF [3], XML [4], CSV [5] and JSON [6]), analysing our results both syntactically (using the grammar proposed in [2] and parsing with ANTLR [7]) and semantically. We find that the schemas we produce accurately reflect the original datasets.

1 Introduction

Data is important for many applications in todays world.

Data is shared in a wide variety of formats: the World Wide Web Consortium recommends sharing data as RDF graphs, while practitioners adopt varying formats, such as relational (CSV datasets shared on Kaggle and the French public portal data.gouv.fr), XML (used for bibliographic notices on PubMed), JSON (used by the French Parliament, e.g. NosSenateurs.fr) and property graphs (used to share Offshore leaks).

Those who use the datasets need to form an understanding of their content and the structure, i.e. they need a dataset description. A dataset description usually takes the form of documentation or a schema.

This information does not always come packaged with the dataset; writing documentation for a dataset is time consuming for the producer, and schemas are rare when sharing semi-structured datasets (e.g. XML, JSON, RDF and PG).

The ABSTRA system addresses this problem by implementing an approach for abstracting any tabular, tree or graph-structured dataset.

The premise of ABSTRA is that any dataset is composed of entities grouped in collections connected by relationships. ABSTRA implements the following steps to produce an abstraction: (i) it transforms any dataset into a labeled directed graph, based on the representations of each data format, (ii) it groups

the nodes of the graph into collections, using the way each data models encodes similarity between data objects, (iii) it selects the main collections (those which represent a large fragment of the dataset) and identifies the relationships between them, (iv) it classifies each main collection into a semantic class.

The abstractions produced by ABSTRA are stored in the form of a relational database. These offer human users a first idea about a dataset, but for other applications, e.g. data validation and data augmentation, an associated schema is necessary.

Thus, the goal of the current project is to construct a schema given any abstraction. We have settled on the property graph schema detailed in [2] as our representation.

The property graph model is used in a wide array of applications (transportation networks, biological networks, finance) to represent "interconnected multi-labelled data enhanced with properties given by key/value pairs". Its expressiveness and the ongoing efforts to standardise a query language for this model make it a good candidate for our project. Schemas also provide a double-role: on the one hand they can be viewed as having a prescriptive role, allowing for data augmentation, while on the other hand they can viewed as fulfilling a descriptive role, offering a summary of the original datasets.

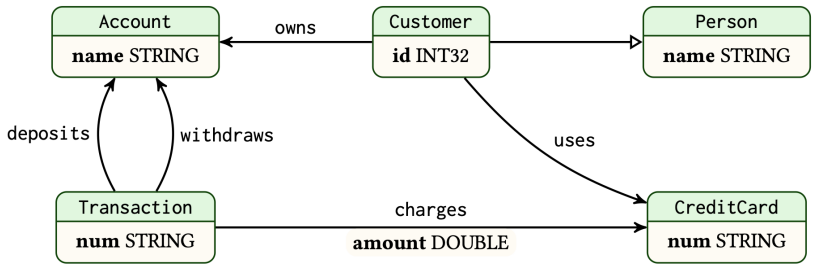


Fig. 1. Visualisation of a property graph schema.

As a summary, a property graph schema is formed by graph types, node types and edge types (see figure 1 and figure 2 for example). Node and edge types may contain one or more labels, and one or more properties with an associated type.

This gives rise to a canonical way of going between an abstraction and a property graph schema: main entities in an abstraction are associated to node types in the schema, entity attributes are associated to properties of node types and relationships between entities in the abstraction are associated to edge types in the schema.

The following section describes the algorithm implemented for creating a schema starting from an abstraction. The code, test abstractions used and gener-

```
CREATE GRAPH TYPE   fraudGraphType STRICT {
  (personType: Person {name STRING}),
  (customerType: personType & Customer {id INT32}),
  (creditCardType: CreditCard {num STRING}),
  (transactionType: Transaction {num STRING}),
  (accountType: Account {id INT32}),
  (:customerType)
    -[ownsType: owns]->
  (:accountType),
  (:customerType)
    -[usesType: uses]->
  (:creditCardType),
  (:transactionType)
    -[chargesType: charges {amount DOUBLE}]->
  (:creditCardType),
  (:transactionType)
    -[activityType: deposits|withdraws]->
  (:accountType)
}
```

Fig. 2. Text representation of a property graph schema.

ated schemas can be found at https://gitlab.inria.fr/nbarret/abstraction-work/-/tree/pg-schema-from-abstraction/pg-schema.

2 Methodology

An abstraction is composed of entities, their attributes and the relationships between entities. One feature of an abstraction is that entities may have a deeply nested structure, i.e. entity attributes that themselves have attributes. We identify two possibilities for representing these attributes: FLATTEN and STRUCTURE.

With FLATTEN attributes are serialised in JSON and reported as node properties, while with STRUCTURE the nested attributes lead the creation of new node types representing the attributes.

One other design decision we considered is what attributes are reported. From the way ABSTRA builds collections it is possible that some entity attributes that appear in the abstraction may not be present in all entries composing a collection. We leverage the fact that the property graph schema allows for node

properties to be denoted as OPTIONAL, and provide the following two options: STRICT and OPEN.

Under STRICT all attributes of an entity are reported and attributes not present in each entry that forms an entity are labeled OPTIONAL, while under OPEN only attributes present in at least `t%` of entries forming an entity are declared in the property graph, for some threshold t.

The above options are reflected in the algorithm by allowing the user to set two corresponding flags. The algorithm is divided into two phases: generating schema node types and generating schema edge types. Before detailing the two phases we explain the manner in which ABSTRA formats its output and how we read and process it.

ABSTRA formats its output as relational databases containing multiple tables, of which two interest us: `REPORTING_DATA_COLLECTIONS` (containing the main entities and their attributes) and `REPORTING_DATA_RELATIONSHIPS` (containing the relationships between main entities). Our code is written in Python and uses the `psycopg` library to query the abstraction databases. As an intermediary representation we use Python dataclasses to store schema nodes, edges and properties.

We follow the following steps to build a property graph schema: (i) we query the table `REPORTING_DATA_COLLECTIONS` for the main entities. Each main entity record contains a field BOUNDARY which records all information about the entity's attributes. We store the main entities as class instances, also recording their type, labels and attributes. ABSTRA assigns unique identifiers to each entity, and we keep a dictionary of identifiers pointing to main entity nodes, (ii) we then recurse through the already formed node types deciding how their attributes are reported, and check for the two flags. Depending on the first flag (STRICT or OPEN) we determine whether to report an attribute as a node type property, and depending on the second flag (FLATTEN or STRUCTURE) we determine whether a nested attribute is serialised or unfolded. For unfolded attributes we also store edges to the nodes that formed them, (iii) we query the table `REPORTING_DATA_RELATIONSHIPS` for the relationships between main entities. Edges are identified by the identifiers of their source and destination. In order to represent information in the schema we use the dictionary constructed in (i) to identify the end nodes of the edges, (iv) we format the constructed nodes and edges so that they comply with the grammar of the property graph schema.

As part of the technical decisions we made in the development of the algorithm we mention: (i) properties in the schema have associated types, which by default we set to "String", (ii) a given attribute may belong to multiple entities in the abstraction. When nested, we decide to represent them as distinct nodes based on their "parent", e.g. the nested attribute "description" belongs to both entities "category" and "item", so when flag two is set to STRUCTURE, we create two new nodes "categorydescriptionType" and "itemdescriptionType", (iii) node labels may be URIs as ABSTRA utilises knowledge bases to map collections. Given an URI we decide to report only the last element in its path

component, (iv) edges constructed between schema nodes and unfolded nested attributes do not have any equivalent in the original abstraction. We decide to associate to them numeric types, e.g. "edge0Type", and meaningful labels, formed by concatenating the source and destination labels, (v) the schema grammar which we used as a reference for the construction of our schemas contains restricted characters and words which may be present in the types and labels we create. We replace restricted characters by underscore and prefix restricted words depending on the context, e.g. "type" is a restricted word, so if we find a property named type we replace it by `property_type`.

The output of our algorithm is a text file containing a property graph schema conforming to the syntax described in [2].

3 Results and analysis

The algorithm described in section 2 automatically produces a schema given an abstraction. We tested our implementation on five abstractions originating from datasets formatted in RDF, CSV, JSON and XML. We evaluate the resulting schemas both syntactically and semantically.

The five datasets and their abstractions are: (i) COMPANIES (CSV), (ii) CONFERENCES (RDF), (iii) PUBMED (XML), (iv) RESEARCHERS (JSON), (v) XMARK (XML).

Syntactically we evaluate the produced schemas by checking that they are validated by the grammar described in [2]. We use ANTLR Lab [7] to this end, an online parser generator. Semantically, we checked by hand whether the produced schemas match the original abstractions, answering the question: "Does the schema produce nodes, edges and properties that correspond to the entities, relationships and attributes in the abstraction?". Our results do indeed match the starting abstractions.

We point out the fact that evaluation by hand is reasonable in this context as schemas serve both descriptive and prescriptive roles. In the context of the current project we produce results aimed at technical human users. Thus, our tests are of small size and can be manually checked.

Figure 3 and 4 contain a comparison of an abstraction and the produced schema.

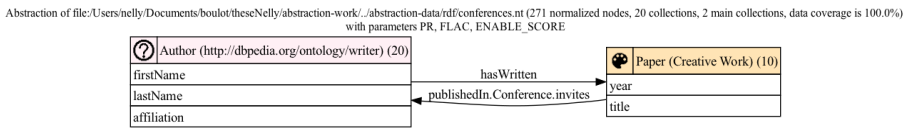


Fig. 3. Abstraction of conferences dataset.

```
CREATE GRAPH TYPE conferencesGraphType STRICT {
    (authorType: Writer {firstName String, lastName String, affiliation String}),
    (paperType: Category_Creative_Work {year String, title String}),
    (:authorType)-[edge0Type: HasWritten]->(:paperType),
    (:paperType)-[edge1Type: PublishedIn_Conference_invites]->(:authorType)
}
```

Fig. 4. Schema generated for abstraction of conferences.

4 Limitations

Currently, our algorithm assigns all properties the generic type "String". This reflects the behaviour of ABSTRA, which does not contain type information. It would be possible to assign more meaningful types, but this would mean leveraging more complex tools, which was outside the scope of this project.

5 Conclusion and perspectives

Property graphs are a powerful data model which are currently undergoing the process of being standardised. Schemas are used to structure the data and can be used in diverse scenarios (prescriptive, descriptive).

Following the proposal in [2], we created an algorithm that automatically produces a property graph schema starting from abstractions generated by the system described in [1]. This conversion addresses the need of abstractions to be computer-readable, and not just human-readable.

We test the produced schemas and see that they match the meaning of their original abstractions.

In the future the schemas could be extended to capture more meaningful information, as described in section 4.

Another more ambitious goal originating from ABSTRA would be to be able to generate common schemas for heterogeneous datasets.

References

1. Nelly Barret, Ioana Manolescu, Prajna Upadhyay. Computing Generic Abstractions from Application Datasets. EDBT 2024 - 27th International Conference on Extending Database Technology, Mar 2024, Paestum, Italy. pp.94-107. hal-04131974v4
2. Renzo Angles, Angela Bonifati, Stefania Dumbrava, George Fletcher, Alastair Green, Jan Hidders, Bei Li, Leonid Libkin, Victor Marsault, Wim Martens, Filip Murlak, Stefan Plantikow, Ognjen Savković, Michael Schmidt, Juan Sequeda, Sławek Staworko, Dominik Tomaszuk, Hannes Voigt, Domagoj Vrgoč, Mingxi Wu, and Dušan Živković. 2023. PG-Schema: Schemas for Property Graphs. Proc. ACM Manag. Data 1, 2, Article 198 (June 2023), 25 pages. https://doi.org/10.1145/3589778
3. https://www.w3.org/TR/2014/REC-rdf11-concepts-20140225/

4. https://www.w3.org/XML/
5. https://www.w3.org/TR/tabular-data-primer/
6. https://json-schema.org/learn/getting-started-step-by-step
7. http://lab.antlr.org