Digital Object Identifier 10.1109/ACCESS.2024.3485685

The Property Graph Data Format (PGDF)

RENZO ANGLES[1,4], SEBASTIÁN FERRADA[2,4], and IGNACIO BURGOS[3,4]
[1]Department of Computer Science, Faculty of Engineering, Universidad de Talca, Curicó, Chile (e-mail: rangles@utalca.cl)
[2]Data & Artificial Intelligence Initiative, Universidad de Chile, Santiago, Chile (e-mail: scferradaa@gmail.com)
[3]Master in Computer Science Program, Department of Computer Science, Universidad de Chile, Santiago, Chile (e-mail: ignacioburgosastete@gmail.com)
[4]IMFD Chile, Santiago, Chile

Corresponding author: Renzo Angles (e-mail: rangles@utalca.cl).

This work was partially funded by ANID Millennium Science Initiative Program, Code ICN17_002, and ANID FONDECYT Chile through grant 1221727.

ABSTRACT Property graphs are popular in both industry and academia due to their versatility in modeling complex data across diverse application domains, ranging from social networks to knowledge graphs. Despite their popularity, there is no standardized data format for storing and exchanging property graphs. This paper introduces PGDF, a text-based data format for property graphs, designed to be both simple and flexible, while remaining expressive and efficient. The simplicity of PGDF comes from its tabular-like structure, where each line in a PGDF file contains a single schema or data declaration. PGDF offers great flexibility by allowing schema and data declarations to be combined in any order. This means that nodes and edges can each have their own distinct properties, providing greater adaptability and customization. The expressiveness of PGDF is defined by its ability to represent a wide range of property graph features. In this article, we describe the syntax and semantics of PGDF, outline methods for converting property graphs stored in multiple CSV files to PGDF and other graph data formats, and present an experimental evaluation comparing PGDF, YARS-PG, GraphML, and JSON-Neo4j. The experiments show that PGDF enables the production of smaller files more quickly compared to other graph data formats.

INDEX TERMS Graph Databases, Property Graphs, Graph Data Formats, PGDF

I. INTRODUCTION

IN recent years, property graphs have emerged as a novel paradigm for representing and analyzing complex relationships within data. Property graphs, which consist of nodes, edges and properties [1], have gained immense prominence not only in academic research [2], [3], [4] but also in industry [5], [6]. Their ability to model intricate connections and attributes in a wide range of applications, from social networks [7], [8] to knowledge graphs [9], has made them an indispensable tool for data management [10], analytics [11], and knowledge discovery [12].

As the demand for graph data utilization continues to grow, a noteworthy challenge has emerged: the lack of a standardized and comprehensive data format for storing and exchanging property graphs. Traditional alternatives like CSV (Comma-Separated Values) or even the more structured GraphSON and GraphML formats [13], while suitable for certain use cases, can fall short in capturing the rich and nuanced features of property graphs [1]; for instance, by not explicitly allowing multi-valued properties (e.g., GraphML [13] is XML-based, and even though XML allows repeating tags to encode multiple values for a property, GraphML does not support this as the property names are encoded in the tag attribute `key`, which must be unique within an edge or node).

In response to the aforementioned issues, we have created PGDF, a data format for property graphs which was designed to satisfy the following characteristics: a) Simplicity: PGDF follows an intuitive tabular-like structure such that each line in a PGDF file contains a single schema or data declaration. b) Flexibility: the schema and data declarations can be combined in any order, allowing nodes and edges with distinct properties. c) Expressiveness: PGDF is able to represent a wide variety of property graph features such as multiple labels, multi-valued properties and edge IDs.

Moreover, PGDF offers the following advantages: PGDF has the potential to be generic, unlike JSON-based serializations which are vendor-specific (i.e., a JSON file obtained with Neo4j may not be imported directly in other systems); PGDF can be used to serialize property graphs obtained from popular graph database systems, including Neo4j, Amazon Neptune and TigerGraph; a property graph serialized as multiple CSV files can be stored in a single PGDF file; and, PGDF produces smaller files than other serializations such as GraphSON, GraphML and YARS-PG.

In this article we introduce PGDF, study transformation methods, and present an empirical evaluation. This paper is

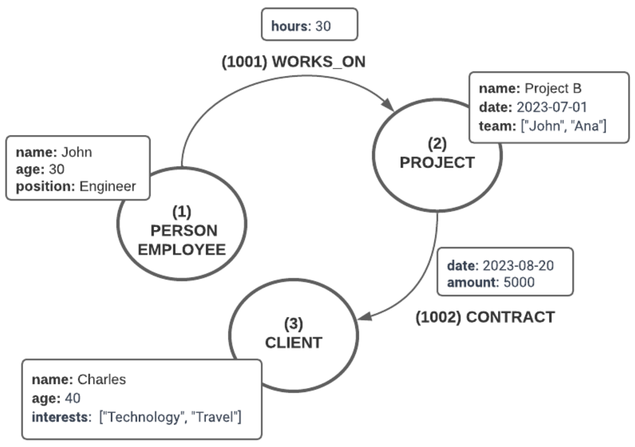


FIGURE 1: Example of a property graph.

organized as follows: Section II contains a formal definition of the property graph data model; Section III includes a review of current data formats for property graphs; Section IV contains the syntax and semantics of PGDF, and an algorithm for producing a PGDF file; Section V includes a method to convert property graphs from CSV to PGDF, including a use-case example based on the LDBC Social Network Benchmark (LDBC-SNB); Section VI explain methods to convert a CSV-based property graph to other graph formats (i.e. YARS-PG, GraphML, JSON); Section VII presents evidence that PGDF is able to produce smaller files faster than other data formats.

II. PROPERTY GRAPHS

A property graph is a directed multi-graph where the nodes and the edges can have labels and properties (i.e. name-value pairs). Figure 1 shows an example of property graph composed of three nodes and two edges. Node 1 has two labels (`PERSON` and `EMPLOYEE`) and three single-value properties (`name`, `age`, and `position`). Node 2 has label `PROJECT`, two single-valued properties (`name` and `date`) and a multi-valued property (`team`). Node 3 has label `CLIENT`, two single-valued properties (`name` and `age`), and a multi-valued property (`interests`). In addition, there are two relationships between the nodes: edge 1001 with label `WORKS_ON`, and edge 1002 with label `CONTRACT`. Both edges also have properties.

We extend the property graph definition presented in [1], to include node and edge identifiers, as well as directionality for edges. For this, we assume the following countably infinite sets: L (node and edge labels), *Props* (property names), *Vals* (property values), and *VID* and *EID* (nodes and edge identifiers).

Definition 1: A **Property Graph** G is a six-tuple $(V, E, \rho, \lambda, \sigma, \delta)$, where:

- $V \subset VID$ is a finite set of nodes;
- $E \subset EID$ is a finite set of edges;
- $\rho : E \to V \times V$ is a function that associates each edge in E with a pair of nodes, both in V;
- $\lambda : V \cup E \to 2^L$ is a function that associates each node and edge with a set (which may be empty) of labels;
- $\sigma : (V \cup E) \times Props \mapsto 2^{Vals}$ is a partial function that connects nodes and edges with property names and property values;
- $\delta : E \to \{\rightarrow, \leftarrow, \leftrightarrow\}$ is a function that assigns a direction to each edge.

Hence, the property graph shown in Figure 1 can be defined as $G = (V, E, \rho, \lambda, \sigma, \delta)$ where:

- $V = \{1, 2, 3\}$,
- $E = \{1001, 1002\}$,
- $\rho = \{1001 \to (1, 2), 1002 \to (2, 3)\}$,
- $\lambda = \{1 \to \{\texttt{PERSON}, \texttt{EMPLOYEE}\},$
  $2 \to \{\texttt{PROJECT}\},$
  $3 \to \{\texttt{CLIENT}\},$
  $1001 \to \{\texttt{WORKS\_ON}\},$
  $1002 \to \{\texttt{CONTRACT}\}\},$
- $\sigma = \{(1, \texttt{name}) \to \{\texttt{John}\}, (1, \texttt{age}) \to \{\texttt{30}\},$
  $(1, \texttt{position}) \to \{\texttt{Engineer}\},$
  $(2, \texttt{name}) \to \{\texttt{Project B}\},$
  $(2, \texttt{date}) \to \{\texttt{23/07/01}\},$
  $(2, \texttt{team}) \to \{\texttt{John}, \texttt{Ana}\},$
  $(3, \texttt{name}) \to \{\texttt{Charles}\}, (3, \texttt{age}) \to \{\texttt{40}\},$
  $(3, \texttt{interests}) \to \{\texttt{Technology}, \texttt{Travel}\},$
  $(1001, \texttt{hours}) \to \{\texttt{30}\},$
  $(1002, \texttt{date}) \to \{\texttt{2023/08/20}\},$
  $(1002, \texttt{amount}) \to \{\texttt{5000}\}\}$; and
- δ is such that $\delta(1001) = \rightarrow$, and $\delta(1002) = \rightarrow$

Another concept that shall become important later on is the one of the schema for nodes and edges. A property graph schema can be understood in many ways, for instance, as rules defining which kinds of nodes can have which properties and should be connected by a given set of edges [4]. For the purposes of our work, we define the schema in a per node and per edge basis by simply saying that the schema of a node (resp. edge) is the set of properties defined for said node (resp. edge). We define this notion formally as follows.

Definition 2 (Schemas): Let $G = (V, E, \rho, \lambda, \sigma, \delta)$ be a property graph. The **schema of a node** $n \in V$ is the set of property names $\mathsf{sch}(n) = \{p \mid (n, p) \in \mathsf{dom}(\sigma)\}$. Similarly, the **schema of an edge** $e \in E$ is the set of property names $\mathsf{sch}(e) = \{p \mid (e, p) \in \mathsf{dom}(\sigma)\}$. We denote by $\mathsf{sch}_{\mathsf{node}}(G)$ the set of all different node schemas in G (i.e., $\mathsf{sch}_{\mathsf{node}}(G) = \{\mathsf{sch}(n) \mid n \in V\}$); and by $\mathsf{sch}_{\mathsf{edge}}(G)$ the set of all different edge schemas in G (i.e., $\mathsf{sch}_{\mathsf{edge}}(G) = \{\mathsf{sch}(e) \mid e \in E\}$).

For example, the schema of node 1 in Figure 1 is $\mathsf{sch}(1) = \{\mathsf{name}, \mathsf{age}, \mathsf{position}\}$, and the schema of the edge 1001 is $\mathsf{sch}(1001) = \{\mathsf{hours}\}$.

III. GRAPH DATA FORMATS

There are several data formats for serializing graphs, but just some of them support property graphs. In this section we review the syntax and semantics of GraphML, YARS-PG, PG Format, JSON-Neo4j, and GraphSON Tinkerpop.

A. GRAPHML

The Graph Markup Language (GraphML) [13] is a file format to store graphs as XML. GraphML has two sections. The first section is the definition of the property names and datatypes of edges and nodes. Before defining the properties of a node, a special property `labelV` can be used to store node labels. Then, all the node property names are defined. Similarly, the property `labelE` can be defined for edge labels followed for all edge property names. In the second section, each node is defined within a `<node>` tag containing its labels and property values; and then each edge is similarly described within an `<edge>` tag. Properties are defined with `<data>` tags specifying the name in the `key` attribute. Multiple `<data>` tags with the same `key` value are not permitted for a node or edge in order to support multi-valued properties. In Figure 2, we present a GraphML file that serializes the data of the property graph shown in Figure 1, where multi-label and multi-valued properties are truncated.

```
<?xml version="1.0" encoding="utf-8"?>
<GraphML>
 <key id="labelV" for="node"
      attr.name="labelV" attr.type="string"/>
 <key id="name" for="node"
      attr.name="name" attr.type="string"/>
 <key id="age" for="node" attr.name="age"
      attr.type="integer"/>
 <key id="team" for="node" attr.name="team"
      attr.type="string"/>
 <key id="position" for="node" attr.name="position"
      attr.type="string"/>
 <key id="interests" for="node" attr.name="interests"
      attr.type="string"/>
 <key id="labelE" for="edge" attr.name="labelE"
      attr.type="string" />
 <key id="hours" for="edge" attr.name="hours"
      attr.type="integer"/>
 <key id="date" for="edge" attr.name="date"
      attr.type="date"/>
 <key id="amount" for="edge" attr.name="amount"
      attr.type="double"/>
 <graph id="G" edgedefault="directed">
  <node id="1">
    <data key="labelV">PERSON</data>
    <data key="name">John</data>
    <data key="age">30</data>
    <data key="position">Engineer</data>
  </node>
  <node id="2">
    <data key="labelV">PROJECT</data>
    <data key="name">Project B</data>
    <data key="date">2023-07-01</data>
    <data key="team">John</data>
  </node>
  <node id="3">
    <data key="labelV">CLIENT</data>
    <data key="name">Charles</data>
    <data key="age">40</data>
    <data key="interests">Technology</data>
  </node>
  <edge id="1001" source="1" target="32">
    <data key="labelE">WORKS_ON</data>
    <data key="hours">30</data>
  </edge>
  <edge id="1002" source="2" target="13">
    <data key="labelE">CONTRACT</data>
    <data key="date">2023-08-20</data>
    <data key="amount">5000</data>
  </edge>
 </graph>
</GraphML>
```

FIGURE 2: Example of the GraphML format.

B. YARS-PG

YARS-PG is a data format designed to support the publication and exchange of property graphs. There are two versions of YARS-PG which are presented in [14] and [15] respectively. In this article we consider the first version because its syntax is simpler than the one defined in the second version.

In a YARS-PG file, declarations for nodes must be defined first, and then followed by declarations for edges. A node declaration begins with the node ID, followed by square brackets (`[]`) that enclose a list of labels for that node separated by colons (`:`), and ends with curly braces (`{}`) that contain key/value pairs representing the property names and values of the node. An edge declaration begins with the ID of the source node, enclosed in parentheses. This is followed by a hyphen (`-`). Within square brackets, one label for the edge is placed, followed by a space, and then the list of property names and values enclosed in curly braces (similar to the case of nodes). Afterwards, an ASCII arrow is appended (`->`). Finally, the ID of the target node is added in parentheses.

In Figure 3, we present the YARS-PG declarations for the property graph shown in Figure 1, where multi-valued properties are truncated, as YARS-PG does not explicitly provide support for them (e.g., by defining arrays). Note that YARS-PG supports only one label per edge.

```
1[PERSON:EMPLOYEE]:{name: "John", age: "30", position: "Engineer"}
2[PROJECT]:{name: "Project B", date: "2023-07-01", team: "John"}
3[CLIENT]:{name: "Charles", age: "40", interests: "Technology"}
(1)-[WORKS_ON {hours: "30"}]->(2)
(2)-[CONTRACT {date: "2023-08-20", amount: "5000"}]->(3)
```

FIGURE 3: Example of the YARS-PG format.

C. PG FORMAT

PG Format [16] is an exchange format similar to YARS-PG, but with a simpler syntax and allowing multi-valued properties by repeating the key. Each node and each edge are on their own line, with space separated fields for IDs, labels and property/value pairs. In Figure 4 we present the serialization of the property graph shown in Figure 1, where it can be noted that edge identifiers are lost.

D. JSON - NEO4J

The Javascript Object Notation (JSON) can also be used to serialize property graphs. Neo4j defines their own import/export JSON format, where each node and edge is serialized as an object distinguished by the `type` property. In each object, labels, property names, and values are serialized in the usual JSON way, for which an example can be found in Figure 5 presenting a JSON for Neo4j that serializes the property graph of Figure 1.

```
1 :Person :Employee name:John age:30 position:Engineer
2 :Project name:"Project B" date:2023-07-01 team:John team:Ana
3 :Client name:Charles age:40 interests:Technology interests:Travel
1 -> 2 :Works_on hours:30
2 -> 3 :Contract date:2023-08-20 amount:5000
```

FIGURE 4: Example of the PG Format.

```
[{
    "type": "node",
    "id": "1",
    "labels": ["PERSON", "EMPLOYEE"],
    "properties": {
      "name": "John",
      "age": "30",
      "position": "Engineer"
    },
  },{
    "type": "node",
    "id": "2",
    "labels": ["PROJECT"],
    "properties": {
      "name": "Project B",
      "date": "2023-07-01",
      "team": ["John", "Ana"]
    },
  },{
    "type": "node",
    "id": "3",
    "labels": ["CLIENT"],
    "properties": {
      "name": "Charles",
      "age": "40",
      "interests": ["Technology", "Travel"]
    },
  },{
    "id": "1001",
    "type": "relationship",
    "label": "WORKS_ON",
    "properties": {
      "hours": "30"
    },
    "start": {
      "id": "1"
    },
    "end": {
      "id": "2"
    }
  },{
    "id": "1002",
    "type": "relationship",
    "label": "CONTRACT",
    "properties": {
        "date": "2023-08-20",
        "amount": "5000"
    },
    "start": {
      "id": "2"
    },
    "end": {
      "id": "3"
    }
}]
```

FIGURE 5: Example of the JSON-Neo4j format.

```
[{"id": {"@type":"g:Int64","@value":1},
  "label": "PERSON",
  "outE": {
    "WORKS_ON": [{
        "id": {"@type":"g:Int64","@value":1001},
        "inV": {"@type":"g:Int64","@value":2},
        "properties": {"hours": "30"}}]},
  "properties": {
    "name": [{
        "id": {"@type":"g:Int64","@value":2001},
        "value": "John"}],
    "age": [{
        "id": {"@type":"g:Int64","@value":2002},
        "value": "30"}],
    "position": [{
        "id": {"@type":"g:Int64","@value":2003},
        "value": "Engineer"}]}},{
  "id": {"@type":"g:Int64","@value":2},
  "label": "PROJECT",
  "inE": {
    "WORKS_ON": [{
        "id": {"@type":"g:Int64","@value":1001},
        "outV": {"@type":"g:Int64","@value":1},
        "properties": {"hours": "30"}}]},
  "outE": {
    "CONTRACT": [{
      "id": {"@type":"g:Int64","@value":1002},
      "inV": {"@type":"g:Int64","@value":3},
      "properties": {
        "date": "2023-08-20",
        "amount": "5000"}}]},
  "properties": {
    "name": [{
        "id": {"@type":"g:Int64","@value":2004},
        "value": "Project B"}],
    "date": [{
        "id": {"@type":"g:Int64","@value":2005},
        "value": "2023-07-01"}],
    "team": [{
        "id": {"@type":"g:Int64","@value":2006},
        "value": "John"},{
        "id": {"@type":"g:Int64","@value":2007},
        "value": "Ana"}]}},{
  "id": {"@type":"g:Int64","@value":3},
  "label": "CLIENT",
  "inV": {
    "CONTRACT": [{
      "id": {"@type":"g:Int64","@value":1002},
      "outV": {"@type":"g:Int64","@value":2},
      "properties": {
        "date": "2023-08-20",
        "amount": "5000"}}]},
  "properties": {
    "name": [{
        "id": {"@type":"g:Int64","@value":2008},
        "value": "Charles"}],
    "age": [{
        "id": {"@type":"g:Int64","@value":2009},
        "value": "40"}],
    "interests": [{
        "id": {"@type":"g:Int64","@value":2010},
        "value": "Technology"},{
        "id": {"@type":"g:Int64","@value":2011},
        "value": "Travel"}]}}]
```

FIGURE 6: Example of the GraphSON-TP3 format.

E. GRAPHSON - TINKERPOP3

GraphSON - TinkerPop 3 (TP3) is a data format for serializing property graphs inspired in JSON and aiming to be easy to split and load in distributed systems. In a GraphSON file, each node is serialized as a JSON object that has the id and labels of the node, along with three nested objects. The first nested object contains all edges going out of the node: their ids, labels and properties. The second nested object contains all edges going into the node. The third nested object contains the property names and values of the node. In Figure 6, we present a GraphSON file that represents the property graph shown in Figure 1, where multi-labels are truncated, as these are not explicitly supported.

F. ANALYSIS OF DATA FORMATS

Table 1 shows a comparison of the data formats described above. We considered different features that can appear in property graphs and state if they are explicitly supported or not. It is evident that none of the formats supports all the listed features, with JSON-Neo4j and PG Format being the most complete. It can be argued that some of the formats can be used in such a way that they support some of the missing features, but we report the explicitly supported features as stated in the papers or documentation of each data format.

IV. THE PROPERTY GRAPH DATA FORMAT (PGDF)

In this section we present PGDF, a data format designed to be simple, flexible, expressive and efficient. PGDF is inspired by the simplicity of the CSV data format, but providing flexibility to accommodate nodes and edges having different schemas. PGDF allows to express all the features of the property graph model, including multiple labels for nodes and edges, and multi-valued properties.

A PGDF file consists of schema declarations and data declarations both for nodes and edges. A schema declaration defines the structure of a specific group of nodes or edges. A data declaration defines the data for a node or edge. Note that a data declaration must strictly follow the structure of the schema declaration defined previously.

The components of a schema declaration are shown in Figure 7a. The schema declaration for a node begins with two reserved keys, `@id` and `@label`, which are separated by a pipe symbol (`|`). Following this, the user-defined property names are listed, also separated by pipes. The schema declaration for an edge is similar to the one for node, but allowing three additional keys: `@dir`, `@out` and `@in`. These attributes are used to indicate the type of the edge (directed or undirected), the source node, and the target node, respectively.

The components of a data declaration are shown in Figure 7b. A data declaration starts with the identifier (ID) of the node or edge, being optional for edges. Then, the labels for the node or edge must be provided either as a single label in the form of a String, or as multiple labels in the form of an Array. For a node, the data declaration ends with its property values. A property value can either be a single-value String or a multi-value Array. For an edge, the next step is to specify its type (`T` for directed, or `F` for undirected), followed by the ID of the source node and the ID of the target node. The declaration ends with the properties of the edge. As defined for a schema declaration, the components of a data declaration are separated by a pipe symbol.

The process to create a PGDF file from a property graph G (as defined in Definition 1) is presented in Algorithm 1. First, it iterates through all different node schemas in G (line 2) and generates a schema declaration (lines 3–7). Then, the algorithm iterates through the nodes in V, such that they have the same schema (line 8). Then, for each such node, the algorithm writes its data declaration consisting of id, labels and property values (lines 9–17). For labels and property values, we use the *asArray* function, which has three possible cases: first, if an empty set is given, the *asArray* returns the empty string; if a singleton set is given, *asArray* returns the string that codifies the single element; if *asArray* receives a larger set, it returns the strings of each element of the set separated by commas. Then, the same process is repeated for the edges (lines 20–49), where the only difference is that edges have the attributes `@dir`, `@in` and `@out`.

In Figure 8, we show the content of the PGDF file obtained for the property graph shown in Figure 1. As it can be noticed, PGDF uses the pipe character (`|`) to separate fields, and the comma character (`,`) to separate multiple values for the same field. A problem arises in practice when the values of the property graph contain such characters. A solution to this problem is to allow users to define a quotation character, such that delimiters found between a pair of quotation characters are ignored. This is common practice in the parsing of character-separated formats.

It can be seen that Algorithm 1 can be run in time proportional to $|V| + |E|$, by first partitioning the nodes and edges with respect to their schema. In practice, it can be expected that nodes and edges with the same schema are stored in the same CSV files and are, therefore, already partitioned. However, we can see here a trade off between expected file size and execution time. Partitioning nodes and edges according to schema ensures that there is only one schema declaration per schema, which is the minimum possible number. However, since PGDF allows a new schema declaration at any moment, each node and each edge can be serialized in just one pass. The "current" schema can be kept, and if the next node or edge fits with the current schema, then a data declaration for the node or edge's information is added. If the next node or edge does not fit with the current schema, then a schema declaration is added with the schema of the node or edge, and it becomes the new current schema. In the worst case for file size, each time a node or edge is visited, the current schema changes, thus there is a schema declaration for each data declaration. However, we expect this case to be rare, and we discuss this later.

V. CONVERTING CSV TO PGDF

CSV (Comma Separated Values) is a file format which is very popular in data management. Several open datasets are distributed on the Web as CSV (e.g., http://kaggle.com). CSV is used by many graph-oriented software to import and export data, including visualization tools and graph-oriented benchmarks (e.g the LDBC-SNB [7]).

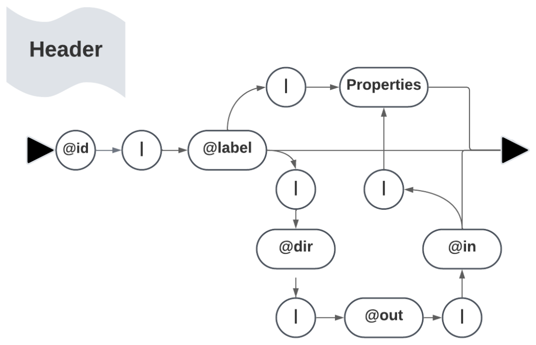
Table 2 shows several graph database systems and the different data formats they support for importing graph data. It can be seen that among the most widely supported formats are CSV and JSON. Between CSV and JSON, we note that CSV is usually the most portable format, as the same files can be used in several systems, whereas JSON is usually particular to each system, as each system requires a certain object structure and mandatory properties. In addition, most database systems also allow to export data as CSV, particularly, relational database systems such as PostgreSQL and MySQL.

For all these reasons, we argue that it is fundamental to provide a simple and automatic way to convert CSV data to PGDF. Moreover, there are two good reasons to use PGDF instead of CSV to store and exchange property graphs: 1)

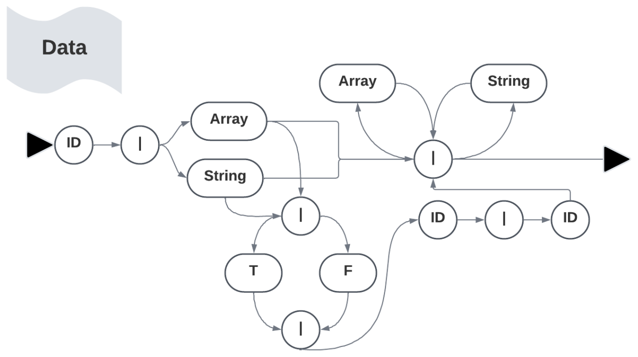
TABLE 1: Property Graph features supported by different data formats.

Feature	YARS-PG	GraphML	JSON-Neo4j	GraphSON	PG Format
Nodes with ID	✓	✓	✓	✓	✓
Edges with ID	✗	✓	✓	✓	✗
Edges without ID	✓	✗	✓	✗	✓
Node and Edge Labels	✓	✓(*)	✓	✓	✓
Multi-labeled Nodes	✓	✗	✓	✗	✓
Multi-labeled Edges	✗	✗	✗	✗	✓
Directed Edges	✓	✓	✓	✓	✓
Undirected Edges	✗	✓	✗	✗	✓
Single-valued Properties	✓	✓	✓	✓	✓
Multi-valued Properties	✗	✗	✓	✓	✓
Null-valued Properties	✓	✓(**)	✓	✓	✓(**)

(*) Supported through special properties. (**) Supported by omission.



(a) The structure of a PGDF schema declaration.



(b) The structure of a PGDF data declaration.

FIGURE 7: Structure of PGDF schema and data declarations.

```
@id|@label|name|age|position
1|PERSON,EMPLOYEE|John|30|Engineer
@id|@label|name|date|team
2|PROJECT|Project B|2023-07-01|John,Ana
@id|@label|name|age|interests
3|CLIENT|Charles|40|Technology,Travel
@id|@label|@dir|@out|@in|hours
1001|WORKS_ON|T|1|2|30
@id|@label|@dir|@out|@in|date|amount
1002|CONTRACT|T|2|3|2023-08-20|5000
```

FIGURE 8: PGDF file for the property graph of Figure 1.

PGDF puts all the information in just one file, facilitating the exchange of the data, as well as potentially reducing cumbersome import commands; and 2) PGDF has been designed to serialize arbitrary property graphs with several features.

In the remainder of this section, we introduce a CSV to PGDF conversion method, a tool that implements this method, and later we showcase a use-case example based upon the LDBC social network benchmark (LDBC-SNB).

A. CONVERSION METHOD

In this section we describe a method for converting a property graphs stored as a set of CSV files to a single PGDF file. When a property graph is exported to CSV, it is often the case that there is a CSV file for each type of node or edge. These "types" usually correspond to the labels of said edges and nodes. Therefore, the input of the CSV-to-PGDF conversion method is a set of CSV files where each node type and edge type is a separate file.

To give some meaning to the input CSV files, we require the creation of a configuration JSON file in which the paths to each individual file are given, such that the user can specify which of these are meant to be nodes and which are meant to be edges, what are the delimiter characters and if the file has a header or not. Furthermore, the user can specify the names of the columns, as they are required by PGDF.

In Figure 9, we show a template of the JSON configuration file. In the figure we can see two root-level objects: `nodes` and `edges`. In the `nodes` object, we list the CSV files that represent nodes where, for each file, the user must assign the following properties:

Algorithm 1: Serialize a Property Graph into PGDF.

```
Data: $G = (V, E, \rho, \lambda, \sigma, \delta)$, a Property Graph.
Result: $out$, a file in the PGDF format
$out \leftarrow$ open(output.pgdf) ;
for $\Sigma \in \mathsf{sch}_{\mathsf{node}}(G)$ do
    $out$.write("@id|@label") ;
    for $p \in \Sigma$ do
        $out$.write("|"); $out$.write($p$) ;
    end
    $out$.newline() ;
    for $n \in \{n \mid n \in V \wedge \mathsf{sch}(n) = \Sigma\}$ do
        $out$.write($n$);
        $out$.write("|") ;
        $out$.write($asArray(\lambda(n))$);
        $out$.write("|") ;
        for $p \in \Sigma$ do
            $out$.write($asArray(\sigma(n, p))$);
            $out$.write("|") ;
        end
        $out$.newline() ;
    end
end
for $\Sigma \in \mathsf{sch}_{\mathsf{edge}}(G)$ do
    $out$.write("@id|@label|@dir|@out|@in") ;
    for $p \in \Sigma$ do
        $out$.write("|"); $out$.write($p$);
    end
    $out$.newline() ;
    for $e \in \{e \mid e \in E \wedge \mathsf{sch}(e) = \Sigma\}$ do
        $out$.write($e$) $out$.write("|");
        $out$.write($asArray(\lambda(e))$);
        $out$.write("|") ;
        if $\delta(e) = \leftrightarrow$ then
            $out$.write("F") ;
        else
            $out$.write("T") ;
        end
        $out$.write("|") ;
        $(n_1, n_2) \leftarrow \rho(e)$ ;
        if $\delta(e) = \leftrightarrow$ or $\delta(e) = \rightarrow$ then
            $out$.write($n_1$); $out$.write("|");
            $out$.write($n_2$) ;
        else
            $out$.write($n_2$); $out$.write("|");
            $out$.write($n_1$) ;
        end
        for $p \in \Sigma$ do
            $out$.write($asArray(\sigma(e, p))$);
            $out$.write("|") ;
        end
    end
end
```

1) `id`, to give an identifier to the CSV file, required to refer to it later;
2) `file`, to indicate the location path (on hard disk) to the corresponding CSV file;
3) `delimiter`, to indicate the character separating the fields in the CSV file;
4) `header`, a boolean value to indicate if the CSV file has a header or not;
5) `labels`, a list of labels to assign to the nodes extracted from the CSV file;
6) `properties`, the list of mandatory PGDF attributes (e.g., `@id`) and property names in the order these appear in the columns of the CSV files.

Then, in the `edges` object, the user must list the CSV files representing edges where, for each file, the user must assign the following properties:

1) `file`, to indicate the location path to the corresponding CSV file;
2) `delimiter`, to indicate the character separating the fields in the CSV file;
3) `header`, a boolean value to indicate if the CSV file has a header or not;
4) `label`, the label to assign to the edges extracted from the CSV file;
5) `dir`, it is `true` or `false` depending on whether the edge is directed or not;
6) `source`, the `id` of the CSV file that contains the ids of the source nodes associated with the edges255 in the CSV file;
7) `target`, the `id` of the CSV file that contains the ids of the target nodes associated with the edges contained in the CSV file;
8) `properties`, the list of mandatory PGDF attributes (e.g., `@in` and `@out`) and property names in the order they appear in the columns of the CSV. The `@id` property is optional for edges.

The template in Figure 9 is used to convert three CSV files to one PGDF file, where two of the CSV files (`nodes1.csv` and `nodes2.csv`) store the nodes and the third file (`edges.csv`) stores the edges.

Then, to convert the set of CSV files to PGDF, we follow Algorithm 1, as each CSV file contains only one type of node or edge schema. First, each CSV file containing nodes is serialized to PGDF. The schema of each node schema is created using the information in its respective JSON object as defined by the JSON configuration file. Then, each PGDF data line is formatted according to the information of each line in the CSV file. We repeat a similar process for edges.

B. CONVERSION TOOL

The CSV-to-PGDF conversion method described above was in Java, and we encapsulated it as a Java Archive (JAR) file, which serves as the distribution format for our tool called `CSV2PGDF`. The tool is publicly available at the following repository: https://github.com/dbgutalca/pgdf.

TABLE 2: Data formats (X-axis) supported by current graph database systems (Y-axis).

GDBMS	GraphML	GraphSON T2	GraphSON T3	CSV	JSON	PG Format
Neo4j	✓	✓	✓	✓	✓	✓
ArangoDB				✓	✓	
OrientDB				✓	✓	
GraphDB				✓	✓	
Amazon Neptune	✓	✓	✓	✓	✓	✓
JanusGraph	✓	✓	✓		✓	
TigerGraph				✓	✓	
Stardog				✓	✓	
Dgraph					✓	
AllegroGraph				✓	✓	
Blazegraph					✓	
TypeDB				✓		
Memgraph				✓	✓	
HugeGraph	✓	✓	✓	✓	✓	
Sparksee	✓			✓		
TerminusDB				✓		
ArcadeDB	✓	✓	✓	✓	✓	
NebulaGraph				✓		
VelocityDB				✓	✓	

```
{"nodes":
 [{
   "id": 1,
   "file": "/path/to/nodes1.csv",
   "delimiter": ",",
   "header": "true",
   "labels": [ "Label1" ],
   "properties": ["@id", "property1", "property2" ]
  },{
   "id": 2,
   "file": "/path/to/nodes2.csv",
   "delimiter": ",",
   "header": "true",
   "labels": [ "Label2" ],
   "properties": ["@id", "property3", "property4"]
  }],
 "edges":
 [{
   "file": "/path/to/edges.csv",
   "delimiter": ",",
   "header": "true",
   "label": "Label3",
   "dir": "true",
   "source": 1,
   "target": 2,
   "properties": ["@id", "@out", "@in", "property5"]
  }]}
```

FIGURE 9: Template of the JSON configuration file required to convert CSV files to PGDF.

The `CSV2PGDF` tool can be executed (e.g. in a Linux command line) by using an instruction of the form:

```
java -jar CSVConverter pgdf /path/to/JSON
          /path/to/output/folder
```

The `CSV2PGDF` tool requires two parameters: the path to the JSON configuration file, and the path to where the user wishes to store the resulting PGDF file. Notice that the paths to the individual CSV files are read from the JSON configuration file.

C. USE-CASE

We now present a use-case example that illustrates how graph data can be converted from its original CSV format to PGDF. To do this, we use the data generator of the LDBC Social Network Benchmark [7], which enables the creation of synthetic property graphs using various scale factors. A generated graph has eight types of nodes: `Comment`, `Forum`, `Organization`, `Person`, `Place`, `Post`, `Tag`, and `TagClass`. These nodes are connected through 23 different kinds of edges, which connect nodes of specific types. For example, the edges of type `hasModerator` connect `Forum` nodes with `Person` nodes. A full description of all types of nodes, edges and property names can be found in [7].

The SNB data generator outputs a collection of CSV files. Each CSV file includes data pertaining to either a node type or an edge type. To convert the multiple CSV files to PGDF using `CSV2PGDF` conversion tool, we need to define the JSON configuration file, of which we present an extract in Figure 10. In said figure, we only show nodes of types `Person`, `Comment`, and `Post`, with edges of type `likes` connecting `Person` with `Post` and `Person` with `Comment`

TABLE 3: Information about the LDBC-SNB property graphs used in this article.

Name	Scale Factor	CSV Size	PGDF size	Difference
G1	SF 0.1	62.0 MB	80.1 MB	29.19%
G2	SF 0.3	188.4 MB	242.8 MB	28.87%
G3	SF 1	717.3 MB	918.5 MB	28.04%
G4	SF 3	2.14 GB	2.73 GB	27.57%
G5	SF 10	7.20 GB	9.16 GB	27.22%
G6	SF 30	22.61 GB	28.54 GB	26.22%

```
{"nodes":
[{
  "id": 1,
  "file": "/sf1/person_0_0.csv",
  "delimiter": "|",
  "header": "true",
  "labels": ["Person"],
  "properties": ["@id", "firstName", "lastName"]
 },{
  "id": 2,
  "file": "/sf1/comment_0_0.csv",
  "delimiter": "|",
  "header": "true",
  "labels": ["Comment"],
  "properties": ["@id", "creationDate",
  "browserUsed", "content"]
 },{
  "id": 3,
  "file": "/sf1/post_0_0.csv",
  "delimiter": "|",
  "header": "true",
  "labels": ["Post"],
  "properties": ["@id", "browserUsed",
  "language", "content"]
 }],
"edges":
[{
  "file": "/sf1/person_likes_post_0_0.csv",
  "delimiter": "|",
  "header": "true",
  "label": "likes",
  "dir": "true",
  "source": 1,
  "target": 3,
  "properties": ["@out", "@in", "creationDate"]
 },{
  "file": "/sf1/person_likes_comment_0_0.csv",
  "delimiter": "|",
  "header": "true",
  "label": "likes",
  "dir": "true",
  "source": 1,
  "target": 2,
  "properties": ["@out", "@in", "creationDate"]
 },{
  "file": "/sf1/Edges/comment_hasCreator_person_0_0.csv",
  "delimiter": "|",
  "header": "true",
  "label": "hasCreator",
  "dir": "true,
  "source": 2,
  "target": 1,
  "properties": ["@out", "@in"]
}]}
```

FIGURE 10: Extract of the JSON configuration file to produce PGDF files from data produced with the data generator of the LDBC Social Network Benchmark.

nodes, and edges of type `hasCreator` connecting `Comment` and `Person` nodes. The complete JSON configuration file required to convert all the CSV files produced by the LDBC-SNB generator can be found in the following URL: https://github.com/dbgutalca/pgdf/blob/main/config0.1.json.

Using the `CSV2PGDF` conversion tool, and the aforementioned JSON configuration file, we can convert the set of CSV files into a single PGDF file. For this experiment, we consider the LDBC-SNB files for several scale factors. In Table 3, we present a comparison between the sum of the sizes of all 31 CSV files of the LDBC-SNB at different scale factors and the resulting PGDF file.[1] It can be seen that PGDF files are in average $\sim 27.8\%$ larger than the CSV files, which is mostly due to the repetition of the labels of the nodes and edges in each line of the file. In Section VII, we show that PGDF is still more convenient than other data formats for property graphs, as PGDF is easier to generate and more compact than the alternatives.

VI. CONVERTING PROPERTY GRAPHS TO OTHER GRAPH DATA FORMATS

In this section, we discuss the serialization of property graphs to the other graph data formats showcased in this paper (GraphML, JSON, GraphSON, and YARS-PG). To do this, we define two conversion methods: one based on main memory (RAM), and one based on secondary memory (hard disk). In the following, we describe these conversion methods and the tools implemented for them.

A. MAIN MEMORY-BASED CONVERSION

The input of this method is a set of in-memory objects that model the nodes and edges of the property graph, along with their labels and properties. This representation is then serialized in the destination data format. This method is most suitable for smaller graphs that can fit in memory.

A property graph can be serialized in two ways. The space-efficient way requires to partition the nodes and edges according to the schema, and then each group can be serialized under only one schema line. However, partitioning can be expensive in practice, thus a second conversion method can be considered. We visit each node and edge once, if the current node or edge has the same schema as the previously processed one, then the node or edge is serialized. If the current node or edge has a different schema, then a new schema declaration is introduced and then the node or edge is serialized. We call the first method PGDF-srt (sorted), and the second PGDF-unsrt (unsorted).

Serialization to YARS-PG is very direct, as each node and each edge, along with their labels and properties, are serialized as a line. To produce Neo4j compliant JSON, we follow a process as defined for YARS-PG. For this, we add a JSON object for each node and edge. For nodes, we use the property `"type": "node"` in the object; and for edges, we use `"type": "relationship"`, and add nested JSON objects for the source and target node ids of the edge.

[1] We converted the size reported by the OS in bytes to MB and GB dividing by 1024. The machine used for these experiments is described in Section VII.

For GraphML, we first need all the property names in the graph. So we iterate through all nodes and edges, and add a `key` tag in the GraphML file for each unique property name. Then, for each node (resp. edge), we complete a `node` (resp. `edge`) tag including the information in the line.

To serialize the graph in GraphSON, we need to keep track of all in-going and out-going edges of each node. In this way, we can serialize each node as a GraphSON object, according to the definition in Section III-E. This suggests that the process of creating a GraphSON file is expensive.

B. DISK CONVERSION METHOD

We also define a procedure to convert a property graph stored in several CSV files to the other data formats. The process to convert CSV to PGDF was presented above, when a JSON configuration file (see Figure 9) is required to define some conversion parameters.

To convert CSV to YARS-PG, we first go through all the files containing nodes. For each file, we go through each line, and using the JSON configuration, we craft the serialization of each node extracting the id, labels and property names and property values from both the JSON configuration and the current CSV line. For edges, we follow the same process.

A similar method can be followed to generate JSON, where a node or relationship object is created for each line in each node/edge CSV file, considering the JSON configuration file.

To serialize the CSV files into GraphML, we first get all the distinct node and edge property names from the JSON configuration file to create the preamble `<key>` tags. Then, we create the `<node>` and `<edge>` objects similarly as we do for PGDF, JSON and YARS-PG.

Finally, converting CSV to GraphSON is less direct than to the other data formats. For each line of each CSV file containing nodes, all files containing edges must be read to get the node's in-going and out-going edges. We make use of the `id` attribute in the JSON configuration file, thus, we read only the edge files that declare as `source` or `target` the `id` of the node being processed.

Conversion from CSV to PGDF, YARS-PG, JSON, and GraphML can be done in $O(|V| + |E|)$ time, whereas to GraphSON requires $O(|V| \cdot |E|)$ time.

C. IMPLEMENTATION

The conversion methods described above were implemented as a Java application and distribute as a JAR file. The code can be found in the GitHub repository of the paper. To execute the tool, we shall use a instruction of the form:

```
java -jar CSVConverter [--memory]
    graphml|json|graphson|yarspg
    /path/to/JSON/configuration
    /path/to/destination/folder
```

where the user must include the desired output format (GraphML, JSON-Neo4j, GraphSON or YARS-PG), the path to the JSON configuration file, and the path to the desired out-put file. The optional parameter `-memory` allows to activate the in-memory conversion.

TABLE 4: File size of LDBC-SNB property graphs generated with different scale factors and serialized in different data formats (PGDF, YARS-PG, GraphML and JSON-Neo4j).

	PGDF	YARS-PG	GraphML	JSON-Neo4j
G1	80.1 MB	117.0 MB	286.0 MB	382.4 MB
G2	242.8 MB	353.0 MB	865.7 MB	1167.7 MB
G3	918.5 MB	1323.0 MB	3233.2 MB	4376.0 MB
G4	2.73 GB	3.92 GB	9.59 GB	13.01 GB
G5	9.16 GB	13.09 GB	32.04 GB	43.52 GB
G6	28.54 GB	40.45 GB	98.37 GB	133.61 GB

VII. EXPERIMENTAL EVALUATION

In this section, we compare PGDF with other graph data formats in terms of output size and runtime. For this experimental evaluation, we use data produced with the data generator of the LDBC Social Network Benchmark (LDBC-SNB) [7]. We chose this generator as there are not many real graphs available, and it allows to generate graphs of different sizes, presenting rich connections. Some real graphs, like the Panama Papers dataset, can also be seen as relational data, as there is not a rich connection dynamic among the edges of the graph (i.e., its topology is similar to a tree).

In Table 3, we present information about the graphs used in the experiments, including their scale factor and the size of the generated CSV files. The experiments were run in two Google Cloud virtual machines with different number of CPUs and RAM. Each machine has a Debian GNU/Linux 11 OS with 2-core Intel(R) Xeon(R) CPUs @ 2.20GHz, and 500 GB SSD. The first machine, which we call M1, has 2 CPUs and 8GB RAM. The second machine, M2, has 4 CPUs and 16 GB RAM. To compile and run the code, we used OpenJDK 17.0.9 and Maven 3.6.3.

A. COMPARISON OF FILE SIZE

First, we compare the sizes of the files produced for PGDF, YARS-PG, JSON and GraphML. We do not report results for GraphSON serialization, as it is excessively time-consuming to produce. Indeed, an attempt to generate GraphSON from G1 was executed for three hours and still did not finish.

The sizes of the files generated for the compared graph formats are presented in Table 4. Additionally, we have also depicted these results in the chart displayed in Figure 11, where the Y-axis is in log scale. In both, Table 4 and Figure 11, we can see that PGDF always uses less disk space than the other data formats. YARS-PG performs slightly worse than PGDF, as this format introduces ASCII art characters for edges, as well as a JSON-like representation of the property values, all of which use more characters, and repeat property names in each line. Both GraphML and JSON perform much worse, as these formats require several delimiter and quotation characters which are not used by PGDF nor YARS-PG.

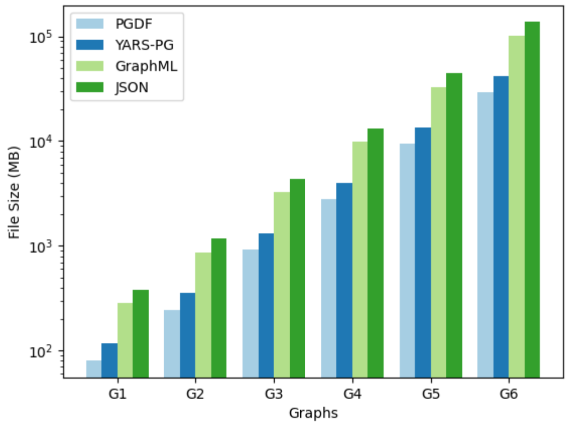


FIGURE 11: The sizes of the files per graph per format.

B. COMPARISON OF CONVERSION TIME

Next, we evaluate the time required to convert property graphs stored in different manners to each data format: PGDF, YARS-PG, JSON-Neo4j, and GraphML. We particularly focus on the memory-based and disk-based conversion methods defined above. We apply these conversion methods to the property graphs G1–6, presented in Table 3. For each conversion method, data format, graph, and machine we report the execution time. Conversion was performed using our CSVConverter tool.

For the memory-based conversion, we first read the CSV files of G1–6 and load them to memory using Java objects. Then, serializations of the in-memory graph are created according to the rules of each format. In Table 5, we present the execution time for each machine, graph and data format. These times are also presented in Figures 12a and 12b for machines M1 and M2, respectively (the Y-axis are in log scale). Considering M1, we can see that only G1 and G2 could be converted before Java runs out of heap space. With M2, we were able to convert G3. It can be seen that PGDF is faster to produce than the alternatives, taking around 70% of the time required to produce YARS-PG. Note that we used the PGDF-unsrt strategy for this conversion. In terms of file size, files produced with PGDF-unsrt are smaller than the ones produced with YARS-PG, despite PGDF-unsrt producing repeated schema declarations. The file sizes obtained with PGDF-unsrt are 89 MB for G1, 270 MB for G2, and 1019 MB for G3.

In the disk-based conversion case, we directly convert the CSV files to the considered data formats. The resulting conversion times (in seconds) are shown in Table 6. These times are also presented in Figures 13a and 13b, where the Y-axis is in log scale. It can be noticed that PGDF is always faster to produce the output. This is because the fields of the CSV file, after being separated, do not require much post-processing to be re-written to PGDF. In contrast, YARS-PG, GraphML and JSON require the combination of each CSV line with the fields declared in the JSON configuration file, which is not needed for PGDF. Furthermore, GraphML takes extra time, as it needs to create the `key` tags at the start of the file with the different property names present in the graph. There is not a significant difference in execution time between the two machines used for the experiment, probably because the most time consuming part of the conversion are I/O operations.

TABLE 5: Execution times of the memory-based conversion.

Machine	Graph	PGDF	YARS-PG	GraphML	JSON-Neo4j
M1	G1	0.231	0.299	0.542	0.409
	G2	0.777	1.118	2.154	1.670
	G3	–	–	–	–
M2	G1	0.183	0.256	0.508	0.377
	G2	0.523	0.744	1.648	1.160
	G3	1.683	2.482	4.708	3.717

TABLE 6: The execution times for the disk-based conversion.

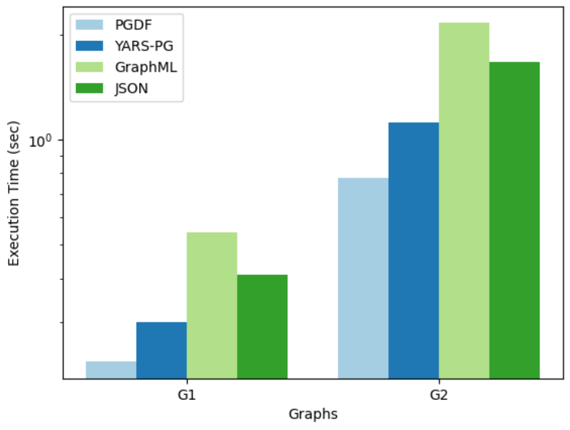
Machine	Graph	PGDF	YARS-PG	GraphML	JSON-Neo4j
M1	G1	0.282	0.389	0.501	0.503
	G2	0.525	0.918	1.276	1.075
	G3	1.373	3.089	4.543	3.750
	G4	3.815	9.305	12.723	10.245
	G5	12.574	31.488	45.631	36.974
	G6	37.553	90.549	132.113	108.283
M2	G1	0.190	0.312	0.430	0.367
	G2	0.392	0.848	1.211	1.025
	G3	1.219	2.993	5.103	3.583
	G4	3.564	8.994	13.151	10.821
	G5	14.177	30.986	45.800	36.860
	G6	39.412	96.147	138.573	111.307

VIII. CONCLUSION

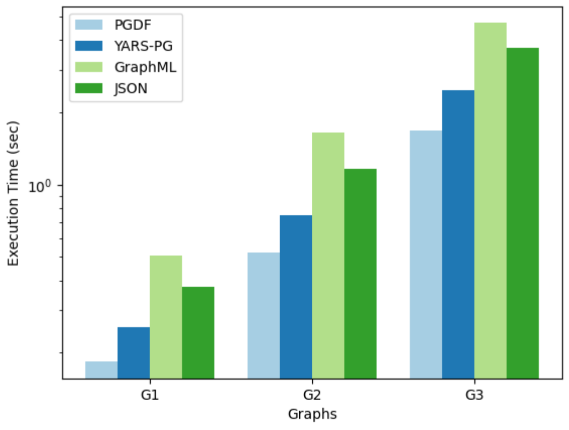
In this paper, we presented PGDF, a text-based data format for serializing property graphs. We showed that PGDF is simple, flexible, expressive, and efficient. We described an algorithm for serializing any property graph to PGDF and a Java tool that implements this algorithm. Furthermore, we defined and implemented various conversion methods from property graphs to YARS-PG, GraphML, and JSON-Neo4j. Our experimental evaluation shows that PGDF uses less disk space and is much faster in producing output than the other formats."

Declaration of generative AI and AI-assisted technologies in the writing process

During the preparation of this work the author(s) used AI-PRO Grammar AI in order to improve language and readability. After using this tool/service, the author(s) reviewed and edited the content as needed and take(s) full responsibility for the content of the publication.

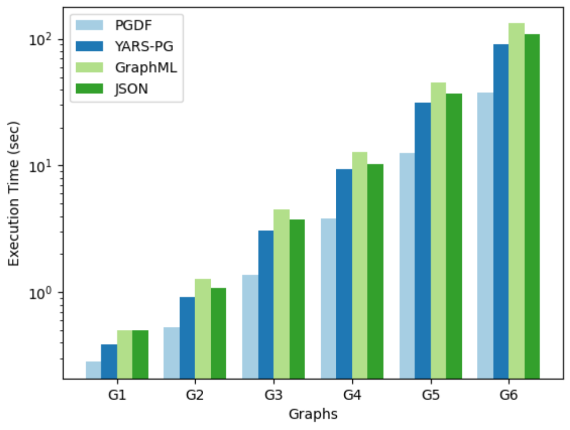


(a) Machine M1.

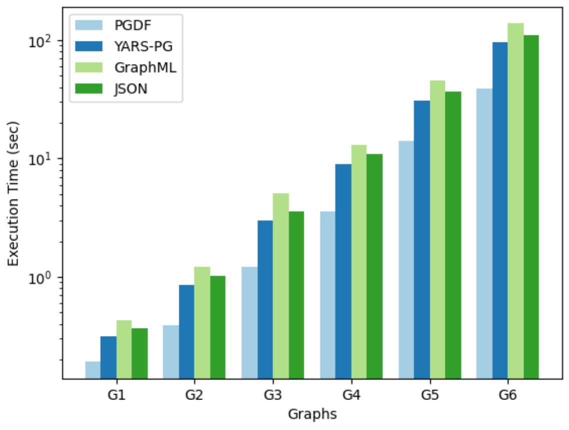


(b) Machine M2.

FIGURE 12: The execution times for the memory-based conversion



(a) Machine M1.



(b) Machine M2.

FIGURE 13: The execution times for the disk-based conversion

REFERENCES

[1] R. Angles, "The Property Graph Database Model," in *Proc. Alberto Mendelzon International Workshop on Foundations of Data Management (AMW)*, vol. 2100. CEUR Workshop Proceedings, 2018.

[2] S. Sakr, A. Bonifati, H. Voigt, A. Iosup, K. Ammar, R. Angles, W. G. Aref, M. Arenas, M. Besta, P. A. Boncz, K. Daudjee, E. D. Valle, S. Dumbrava, O. Hartig, B. Haslhofer, T. Hegeman, J. Hidders, K. Hose, A. Iamnitchi, V. Kalavri, H. Kapp, W. Martens, M. T. Özsu, E. Peukert, S. Plantikow, M. Ragab, M. Ripeanu, S. Salihoglu, C. Schulz, P. Selmer, J. F. Sequeda, J. Shinavier, G. Szárnyas, R. Tommasini, A. Tumeo, A. Uta, A. L. Varbanescu, H. Wu, N. Yakovets, D. Yan, and E. Yoneki, "The future is big graphs: a community view on graph processing systems," *Commun. ACM*, vol. 64, no. 9, pp. 62–71, 2021.

[3] R. Angles, A. Bonifati, S. Dumbrava, G. Fletcher, K. W. Hare, J. Hidders, V. E. Lee, B. Li, L. Libkin, W. Martens, F. Murlak, J. Perryman, O. Savkovic, M. Schmidt, J. F. Sequeda, S. Staworko, and D. Tomaszuk, "PG-Keys: Keys for Property Graphs," in *Proc. of the International Conference on Management of Data (SIGMOD)*. New York, NY, USA: ACM, 2021, pp. 2423–2436.

[4] R. Angles, A. Bonifati, S. Dumbrava, G. Fletcher, A. Green, J. Hidders, B. Li, L. Libkin, V. Marsault, W. Martens, F. Murlak, S. Plantikow, O. Savkovic, M. Schmidt, J. Sequeda, S. Staworko, D. Tomaszuk, H. Voigt, D. Vrgoc, M. Wu, and D. Zivkovic, "PG-Schema: Schemas for Property Graphs," *Proc. ACM Manag. Data*, vol. 1, no. 2, jun 2023.

[5] I. 39075, "Information technology — database languages — GQL," International Organization for Standardization, Geneva, CH, Standard, 2023.

[6] I. 9075-16, "Information technology — database languages SQL — part 16: Property graph queries (SQL/PGQ)," International Organization for Standardization, Geneva, CH, Standard, 2022.

[7] R. Angles, J. B. Antal, A. Averbuch, A. Birler, P. Boncz, M. Búr, O. Erling, A. Gubichev, V. Haprian, M. Kaufmann *et al.*, "The LDBC social network benchmark," *arXiv preprint arXiv:2001.02299*, 2020.

[8] P. Gupta, A. Goel, J. Lin, A. Sharma, D. Wang, and R. Zadeh, "WTF: The Who to Follow Service at Twitter," in *Proc. of the 22nd International Conference on World Wide Web*. New York, NY, USA: ACM, 2013, p. 505–514.

[9] A. Hogan, E. Blomqvist, M. Cochez, C. D'amato, G. D. Melo, C. Gutierrez, S. Kirrane, J. E. L. Gayo, R. Navigli, S. Neumaier, A.-C. N. Ngomo, A. Polleres, S. M. Rashid, A. Rula, L. Schmelzeisen, J. Sequeda, S. Staab, and A. Zimmermann, "Knowledge graphs," *ACM Comput. Surv.*, vol. 54, no. 4, jul 2021.

[10] R. Angles and C. Gutierrez, *An Introduction to Graph Data Management*. Cham: Springer International Publishing, 2018, pp. 1–32.

[11] Y. Xia, I. G. Tanase, L. Nai, W. Tan, Y. Liu, J. Crawford, and C.-Y.

Lin, "Graph analytics and storage," in *Proc. of the IEEE International Conference on Big Data (Big Data)*. IEEE, 2014, pp. 942–951.

[12] C. Wise, V. N. Ioannidis, M. R. Calvo, X. Song, G. Price, N. Kulkarni, R. Brand, P. Bhatia, and G. Karypis, "Covid-19 knowledge graph: Accelerating information retrieval and discovery for scientific literature," 2020. [Online]. Available: https://arxiv.org/abs/2007.12731

[13] U. Brandes, M. Eiglsperger, J. Lerner, and C. Pich, "Graph markup language (GraphML)," in *Handbook of graph drawing and visualization*. CRC press, 2013, pp. 517–542.

[14] D. Tomaszuk, R. Angles, Ł. Szeremeta, K. Litman, and D. Cisterna, "Serialization for property graphs," in *International Conference Beyond Databases, Architectures and Structures (BDAS)*. Cham: Springer, 2019, pp. 57–69.

[15] L. Szeremeta, D. Tomaszuk, and R. Angles, "YARS-PG: Property Graphs Representation for Publication and Exchange," *IEEE Access*, vol. 12, pp. 73 386–73 399, 2024.

[16] H. Chiba, R. Yamanaka, and S. Matsumoto, "Property graph exchange format," 2019. [Online]. Available: https://arxiv.org/abs/1907.03936

RENZO ANGLES is associate professor in the Department of Computer Science at the Universidad de Talca (Chile). He received a degree of Bachelor in Systems Engineering from Universidad Católica de Santa María (Arequipa, Perú), and a Ph.D. degree in Computer Science from Universidad de Chile in 2009. During 2013, he carried out postdoctoral research at the Department of Computer Science, VU University Amsterdam, as part of his participation in the Linked Data Benchmark Council Project. Currently, he participates as researcher in the Millennium Institute for Foundational Research on Data (Chile). His research areas are Data Management (Graph Data Models, Graph Query Languages, Graph Benchmarking) and Bioinformatics (Protein Analysis).

SEBASTIÁN FERRADA is Assistant Professor at the Data & Artificial Intelligence Initiative of Universidad de Chile. He is also a Collaborating Researcher in the Millennium Institute for Foundational Research on Data (Chile). Previously, he worked at the Department of Computer Science at Universidad de Chile and earlier at Linköping University, Sweden. He obtained the Bachelor (2017), the Master (2017) and the PhD (2021) in Computer Science from Universidad de Chile. His research areas are Graph Databases, Knowledge Graphs, Multimedia Databases, and Federated Data Management.

IGNACIO BURGOS received the bachelor's degree in computer science from the Universidad de Talca, Chile, in 2022. Since March 2023, he belongs to the Master in Computer Science Program at the Universidad de Chile. His research areas are Pattern Recognition and Graph Databases.

•••