IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received 11 February 2024, accepted 19 April 2024, date of publication 22 May 2024, date of current version 31 May 2024.

Digital Object Identifier 10.1109/ACCESS.2024.3403924

RESEARCH ARTICLE

YARS-PG: Property Graphs Representation for Publication and Exchange

ŁUKASZ SZEREMETA[1], DOMINIK TOMASZUK[1,2], AND RENZO ANGLES[3]
[1]Department of Computer Science, University of Bialystok, 15-245 Białystok, Poland
[2]Databases and Artificial Intelligence, Technischen Universität Wien, 1040 Vienna, Austria
[3]Department of Computer Science, Faculty of Engineering, Universidad de Talca, Curicó 3340000, Chile

Corresponding author: Łukasz Szeremeta (l.szeremeta@uwb.edu.pl)

The work of Renzo Angles was supported by ANID FONDECYT Chile under Grant 1221727.

ABSTRACT Graph serialization is a critical aspect of advancing graph-oriented systems and applications. Despite the importance of standardized serialization for property graphs, there is a lack of a universal format encompassing all essential features of graph database systems. This study introduces YARS-PG, a simple, extensible, and platform-independent serialization format tailored for property graphs. YARS-PG supports all the features permitted by current property graph-based database systems and is compatible with other graph-oriented databases and tools. We delineate the design requirements of YARS-PG by detailing both functional and non-functional aspects. Besides the basic features of property graph data, YARS-PG supports schema definition, metadata, metaproperties, variables, and graph definitions. Moreover, we discuss extensions of YARS-PG, demonstrating its flexibility through canonicalization techniques. Our comparative analyses with existing formats provide valuable insights, emphasizing the unique strengths that distinguish YARS-PG in the realm of graph data interchange. This paper serves as a definitive guide to YARS-PG, unraveling its complexities and showcasing its potential as a communication protocol, a data storage format, and a messaging specification.

INDEX TERMS Data models, data processing, data structures, database systems, metadata, NoSQL databases, databases.

I. INTRODUCTION

Data serialization is important in data management as it allows for database exchange, systems benchmarking, data visualization, and data presentation. More specifically, data serialization simplifies translation into other data formats, enables automatic data processing, facilitates the comparison of databases (because the same data can be shared between systems), enables the interoperability of heterogeneous databases, and results in a simpler backup method. Therefore, graph serialization is a critical aspect of the advancement of database systems and applications.

In the context of graph data management, there exists no standard data format that encompasses the essential features of graph databases. Hence, this paper endeavors to bridge this gap by introducing YARS-PG, a data format for serializing property graph databases.

Property graph databases present unique challenges for data serialization as they allow diverse and complex data structures. Hence, YARS-PG stands out as a versatile and efficient computer data interchange format specifically designed to facilitate the serialization and deserialization of complex data. This data format is adept at converting intricate objects into streamlined sequences of bits, making it an ideal choice for both persistent data storage and as a wire format for client-service communication. Its cross-platform compatibility ensures seamless data exchange, which is particularly beneficial in networked environments.

The remainder of this article is organized as follows: In Section II, we present the design requirements of YARS-PG, establishing the criteria and parameters that guided its development. In Section III, we formalize the basic concepts associated with property graph databases.

The associate editor coordinating the review of this manuscript and approving it for publication was Vivek Kumar Sehgal.

In Section IV, we describe all the components of YARS-PG. In Section V, we describe the process of exporting a property graph database into a YARS-PG data file. In Section VI, we explore extensions of YARS-PG. In Section VII, we assess the effectiveness and performance of YARS-PG. Finally, in Section VIII, we present conclusions, insights, and future research.

II. DESIGN REQUIREMENTS

In this section, we elaborate on the design requirements for a suitable data format for property graphs. These requirements are compatible with SQL/PGQ [1] and GQL [2], two ISO standards followed by current database technologies.

A. DATA FORMATS FOR PROPERTY GRAPHS

In this section we review the available data formats for serializing property graphs. Such formats are divided into four groups: XML-based, JSON-based, tabular-based, and text-based data formats.

1) XML-BASED DATA FORMATS

In this group are GEXF, GraphML, DotML, DGML and GXL. Graph Exchange XML Format (GEXF) [3] was specifically designed to be used with Gephi, a network analysis and visualization software. GEXF serves as a serialization method for defining intricate network elements like nodes, edges, properties, hierarchies, and associated data. However, it does not support multi-labels for nodes. GraphML [4] is a widely used data format which facilitates properties for nodes and edges, sub-graphs, hierarchical graphs, and hyperedges. However, it does not support null values nor multi-labels. Dot Markup Language (DotML) [5] is a data format developed along with Graphviz, a graph visualization software. Although it supports the serialization of different types of graphs, it has restrictions for encoding property graphs. Directed Graph Markup Language (DGML) [6] supports cyclical and acyclic-directed graphs but with several limitations. Graph Exchange Language (GXL) [7] is used for data interoperability between reverse engineering tools, such as parsers and other tools. It is similar to GraphML and also lacks support for multi-labels.

2) JSON-BASED DATA FORMATS

This group includes to GraphSON TinkerPop 2, GraphSON TinkerPop 3 and JGF. GraphSON is a part of TinkerPop, the open-source graph computing framework, which has its implementations for many databases. In contrast to GraphSON TinkerPop 2 [8], the latest version of this format supports multiple labels for nodes, although these labels must be unique. GraphSON TinkerPop 3 [9] has partial support for defining several values for one key. Both versions handle properties but do not support undirected edges. It is worth noting that GraphSON TinkerPop 3 is not backward compatible. Another JSON-based format, JGF [10], lacks support for properties and multiple labels.

3) TABULAR-BASED DATA FORMATS

In this group are GUESS GDF, Pajek NET, Netdraw VNA and pure CSV. GUESS GDF [11] is related to the GUESS tool used for exploring and visualizing graphs. It has a CSV-like structure and separates blocks of vertex and edge declarations. Its syntax is basic and lacks support for multiple values and properties. Pajek NET [12] supports multiple labels for nodes and undirected edges, but it is not possible to encode properties for edges. Netdraw VNA [13] supports properties and multiple edges with the same label. Unique labels are required for nodes, and column values are separated by whitespaces. Unfortunately, it does not support multi-values and multiple labels. Pure CSV format [14] is supported by current graph database systems (e.g. Neo4j, OrientDB, TigerGraph and Memgraph). It supports tables with columns that can be mapped to different graph structures in different ways. However, its use is system-dependent.

4) TEXT-BASED DATA FORMATS

This group includes to GML, GraphViz DOT, UCINET DL, Tulip TLP, S-Dot, and TGF. Graph Modelling Language (GML) [15] is a structure based on key-value lists that can be nested. Unfortunately, GML does not support multi-values. Graphviz DOT [16] is a data format used in various fields as it allows data collection and stylization of graphs. The drawback of this format is the lack of multigraph support. UCINET DL [17] is a data format based on matrices and lists. It is unable to use multi-values. Tulip TLP [18] has a structure based on round brackets. It allows for collecting data and stylizing the graph. S-Dot [19] is similar to the DotML format. Unfortunately, this format does not support properties. TGF [20] is extremely simple and supports only labels.

B. FUNCTIONAL REQUIREMENTS

In order to define a list of functional requirements (FRs) for YARS-PG, we conducted two tasks: (1) we created a lattice of data models related to property graphs (Figure 1); and (2) we evaluated the property graph features supported by current graph database systems (Table 1). Based on these inputs, we defined the following functional requirements:

- FR1 Nodes: The data format must provide a way to represent nodes in a graph. A node represents an entity and must have a unique identifier within the graph.
- FR2 Edges: The data format must provide a way to represent edges in a graph. An edge represents a relationship between nodes, and it could have an identifier.
- FR3 Labels for nodes: The data format must support labels for nodes. A node label provides a way to categorize a node within the graph based on its type or purpose.
- FR4 Labels for edges: The data format must support labels for edges. An edge label provides a way to categorize

a relationship, allowing users to differentiate between different types of connections.

FR5 Properties for nodes: The data format must allow users to attach properties to nodes. A node property has both a name and a value.

FR6 Properties for edges: The format must allow users to attach properties to edges. An edge property has both a name and a value.

FR7 Types of edges. The data format must support directed and undirected edges.

FR8 Types of property values. The data format must support the representation of single-value properties, multi-value properties and null-value-properties.

FR9 Schemas: The data format should support the organization of data as a blueprint of how the data format is constructed.

FR10 Metadata representation: The data format should incorporate a mechanism for embedding metadata in properties. This feature enables additional contextual information about the properties, such as creation date, authorship, and role. For instance, given a node labeled as `PERSON` with a property `birthDate:"1983-03-31"`, it should be possible to annotate this property with metadata indicating that it was added on `2024-01-10` by a user named `editor1` fulfilling the role of `writer`.

FR11 Complex datatypes: The data format should support composite or compound data types. These sophisticated data structures can be assembled from both primitive data types and other composite types. This ability is crucial for representing complex data in a more structured and hierarchical manner.

FR12 Reusing fragments: The data format should define a variable as a named container designated for storing specific data values. This concept is pivotal for assigning and retrieving data efficiently within the format. Additionally, the format must support the capability to be used multiple times and from various data structure segments.

C. NON-FUNCTIONAL REQUIREMENTS

In this section, we outline the non-functional requirements (NFRs) for YARS-PG, i.e. quality attributes that guided the design of the data format. Additionally, we also defined a criterion for measuring each NFR, using metrics that are categorized into two groups: discrete metrics and continuous metrics. Each metric falls within the range of 0 to 1. The evaluation of a data format should be conducted within the context of a property graph data model.

We distinguish four NFRs:

NFR1 Descriptiveness: YARS-PG must support metadata.

NFR2 Well-definedness: YARS-PG must support a formal syntax description.

NFR3 Flexibility: YARS-PG must allow encoding data portions in different ways.

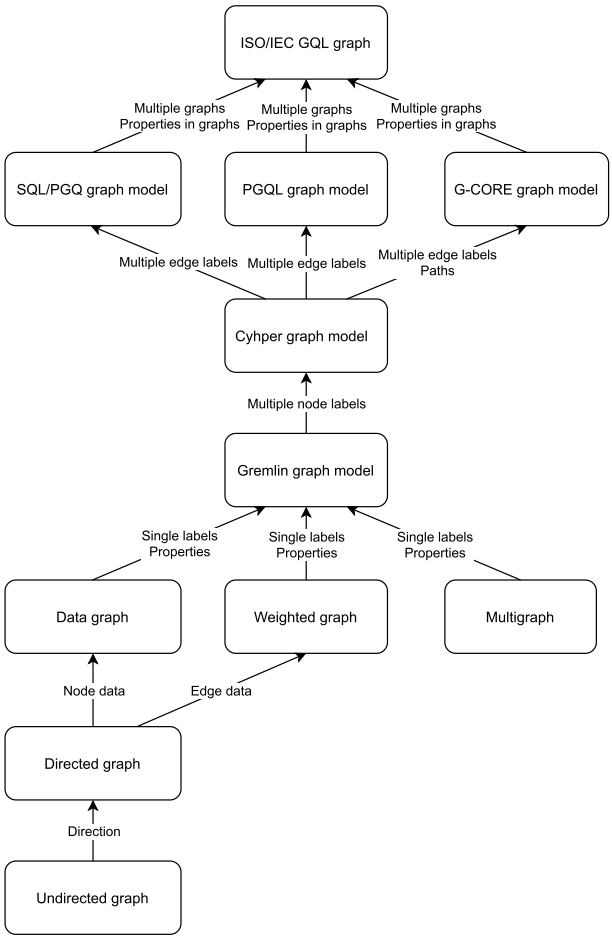

FIGURE 1. Lattice of data models related with property graphs.

NFR4 Expressiveness: YARS-PG must support syntactic elements to express different conceptual elements.

III. PROPERTY GRAPH DATABASES

This section serves as a foundational introduction to the notion of property graph databases. First, we explain basic concepts by using a running example. Then, we present formal definitions for property graphs and property graph schemas.

A. RUNNING EXAMPLE

A property graph is a directed labeled multigraph where nodes and edges could have a set of properties, each represented as a label-value pair. The primary components of a property graph are nodes, edges, and properties. The secondary components are labels (for nodes, edges, and properties) and data types for property values.

Figure 2 presents a graphical representation of a property graph modeling bibliographic information. Nodes are drawn as ellipses, edges as arrows, node labels are drawn inside squares, edge labels occur as floating strings close to the arrows, and properties are drawn as expressions of the form `label:value`. In terms of data modeling, nodes are used to

TABLE 1. Property graph features supported by current graph database systems.

	Node labels			Edge labels			Edges				Properties		
System / Format	Zero	One	Many	Zero	One	Many	Directed	Undirected	Multiple	Duplicated	Mono-value only	Multi-value	Null value
Neo4j	•	•	•		•	•	•		•	•		•	•
Datastax		•			•		•		•	•		•	
OrientDB		•			•		•	•	•	•		•	•
ArangoDB		•			•		•	•	•	•		•	•
JanusGraph	•	•			•		•	•	•	•		•	
Amazon Neptune	•	•	•		•		•		•	•		•	
TigerGraph		•			•		•	•	•			•	
InfiniteGraph		•			•		•	•	•	•		•	•
InfoGrid	•	•	•		•		•	•			•		•
Sparksee		•			•		•	•	•	•	•		•
Memgraph	•	•	•		•		•		•	•		•	•
VelocityDB		•			•		•	•	•	•		•	•
AgensGraph	•		•		•	•	•		•	•		•	
TinkerGraph		•			•		•		•	•		•	
HGraphDB		•			•		•		•	•	•		•
Nebula Graph	•	•	•										
MS Cosmos		•			•		•		•	•		•	•
Oracle Database		•	•		•		•		•	•		•	•
Gremlin		•			•		•		•	•		•	•
Oracle PGQL		•	•		•		•		•	•		•	•
ISO/IEC GQL	•	•	•	•	•	•	•	•	•		•	•	•
ISO/IEC SQL/PGQ		•	•		•		•		•			•	•

represent entities (like authors, bibliographic entries, venues, and journals), edges represent relations between entities, labels can be considered as entity types (e.g., `author`) or relation types (e.g., `has_author`), and properties represent specific attributes for entities and relations (e.g., the `order` of an author in an article).

The `Entry` with the `InProceedings` label is a conference paper with a title beginning with `Serialization for...`, comprises 10 pages, and is associated with the `Graph database` keyword. This paper is part of proceedings titled `BDAS`, dated May 2018. It is authored by John Smith, who is indicated as the first author by the `order` property on the `has_author` edge.

The `Entry` with the `Article` label is a journal article titled `Property Graph...`, also comprising 10 pages, and tagged with the keywords `Query` and `Graph`. This article is published in the `Journal` named `J. DB`, year 2020, volume 30, and has a citation index value (if 4.321) provided by Clarivate Analytics. Alice Brown is the sole author of this article, as indicated by the `order` property on the `has_author` edge.

The graph structure also contains a citation relationship, where the `Article` entry cites the `InProceedings` entry. This relationship is depicted by the `cites` edge connecting the two publication nodes.

B. PROPERTY GRAPH (FORMAL DEFINITION)

Assume that $\mathbf{L}$ is an infinite set of labels, $\mathbf{V}$ is an infinite set of atomic values, and $\mathbf{T}$ is a finite set of data types (e.g., *integer*). Assume that for each value $v \in \mathbf{V}$, the function type(v) : $\mathbf{V} \rightarrow \mathbf{T}$ returns the data type of v. The values in $\mathbf{V}$ will be distinguished as quoted strings. Given a set X, we assume that $\mathrm{SET}^+(X)$ is the set of all finite subsets of X, excluding the empty set.

Definition 1: A property graph is a tuple $G = (g, N, E, P, gp, \sigma, \lambda, \rho, \gamma, \delta)$ where:

1) N is a finite set of identifiers for nodes;
2) E is a finite set of identifiers for edges;
3) P is a finite set of identifiers for properties;
4) It applies that N, E and P are disjoint sets;
5) $g \in \mathbf{L}$ is the label of G;
6) $gp : \mathbf{L} \nrightarrow \mathbf{V}$ is a partial function that defines a set of properties which are specific to G;
7) $\sigma : E \rightarrow (N \times N)$ is a total function that associates each edge with a pair of nodes;
8) $\lambda : (N \cup E) \nrightarrow \mathrm{SET}^+(\mathbf{L})$ is a partial function that associates each node/edge with a non-empty set of labels;
9) $\rho : P \rightarrow \mathbf{L} \times \mathbf{V}$ is a total function that associates each property with a pair label-value.
10) $\gamma : P \times \mathbf{L} \nrightarrow \mathbf{V}$ is a partial function that defines meta-properties for properties;
11) $\delta : P \nrightarrow (N \cup E \cup P)$ is a partial function that associates each property with a node, edge, or property;
12) For each sequence of properties $(p_1, \ldots, p_n)$ where $\delta(p_k) = p_{k-1}$ for $1 < k \leq n$, it applies that $\delta(p_1) \in N \cup E$ and $p_i \neq p_j$.

We will use the term "object" to make reference to any element in the set $N \cup E \cup P$. Given two nodes $n_1, n_2 \in N$ and an edge $e \in E$, such that $\sigma(e) = (n_1, n_2)$, we will say that

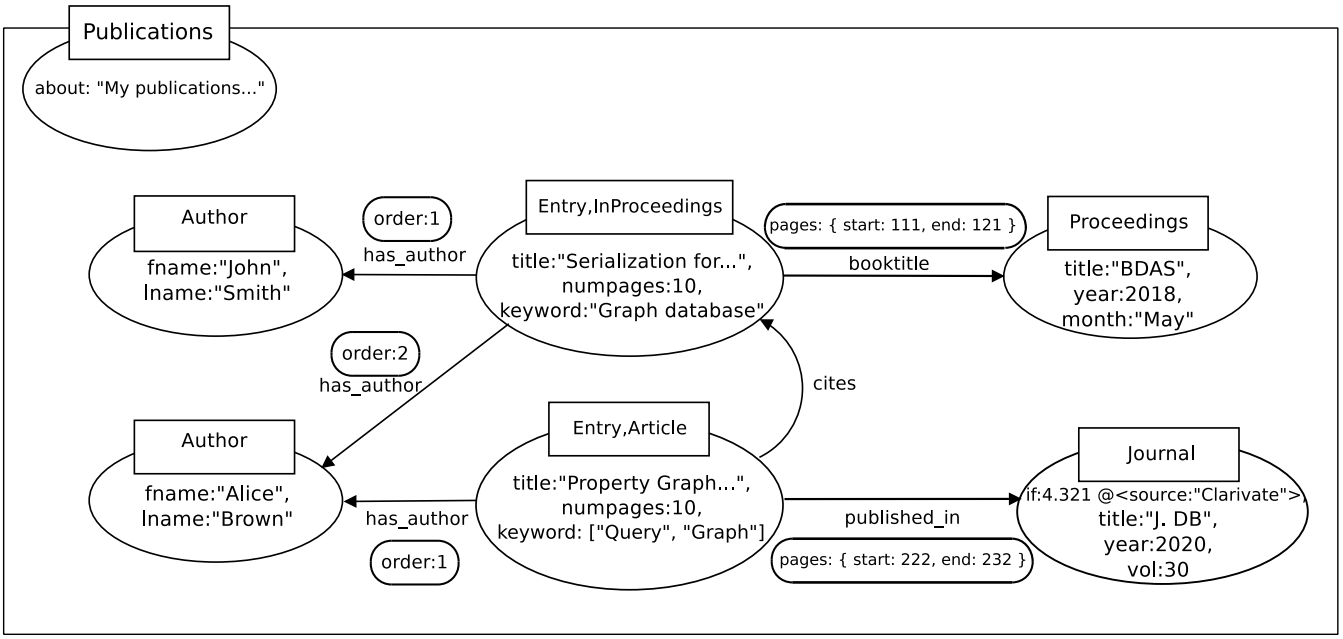

FIGURE 2. Example of property graph representing bibliographic information.

n_1 and n_2 are the "source node" and the "target node" of e respectively. Given a property p with $\delta(p) = o$ and $\rho(p) = (l, v)$, we will use $(o, l) = v_i$ to denote that object o has a property l with value v. Moreover, if there is $\gamma(p, l) = v$ then we will say that l is a "meta-property" of p.

Given two properties p_1, p_2 such that $\delta(p_2) = p_1$, we say that p_2 is a "sub-property" of p_1. Note that the definition of δ allows multiple levels of sub-properties. However, a sub-property belongs to a single property.

Following our formal definition, the property graph shown in Figure 2 could be the tuple $(g, N, E, P, gp, \sigma, \lambda, \rho, \gamma, \delta)$ where:

$g = \texttt{Publications}$,
$N = \{n_1, n_2, n_3, n_4, n_5, n_6\}$,
$E = \{e_1, e_2, e_3, e_4, e_5, e_6\}$,
$P = \{p_0, p_1, p_2, p_3, p_4, p_5, p_6\}$,
$gp(\texttt{about}) = \text{"}\texttt{My publications...}\text{"}$,
$\lambda(n_1) = \{\texttt{Author}\}$,
$\rho(p_1) = (\texttt{fname}, \text{"}\texttt{John}\text{"}), \delta(p_1) = n_1$,
$\rho(p_2) = (\texttt{lname}, \text{"}\texttt{Smith}\text{"}), \delta(p_2) = n_1$,
$\lambda(n_2) = \{\texttt{Entry}, \texttt{InProceedings}\}$,
$\rho(p_3) = (\texttt{title}, \text{"}\texttt{Serialization for ...}\text{"}), \delta(p_3) = n_2$,
$\rho(p_4) = (\texttt{numpages}, \text{"}\texttt{10}\text{"}), \delta(p_4) = n_2$,
$\rho(p_5) = (\texttt{keyword}, \text{"}\texttt{Graph database}\text{"}), \delta(p_5) = n_2$,
$\lambda(n_3) = \{\texttt{Proceedings}\}$,
$\rho(p_6) = (\texttt{title}, \text{"}\texttt{DBAS}\text{"}), \delta(p_6) = n_3$,
$\rho(p_7) = (\texttt{year}, \text{"}\texttt{2018}\text{"}), \delta(p_7) = n_3$,
$\rho(p_8) = (\texttt{month}, \text{"}\texttt{May}\text{"}), \delta(p_8) = n_3$,
$\lambda(n_4) = \{\texttt{Author}\}$,
$\rho(p_9) = (\texttt{fname}, \text{"}\texttt{Alice}\text{"}), \delta(p_9) = n_4$,
$\rho(p_{10}) = (\texttt{lname}, \text{"}\texttt{Brown}\text{"}), \delta(p_{10}) = n_4$,
$\lambda(n_5) = \{\texttt{Entry}, \texttt{Article}\}$,
$\rho(p_{11}) = (\texttt{title}, \text{"}\texttt{Property Graph ...}\text{"}), \delta(p_{11}) = n_5$,
$\rho(p_{12}) = (\texttt{numpages}, \text{"}10\text{"}), \delta(p_{12}) = n_5$,
$\rho(p_{13}) = (\texttt{keyword}, [\text{"}\texttt{Query}\text{"}, \text{"}\texttt{Graph}\text{"}]), \delta(p_{13}) = n_5$,
$\lambda(n_6) = \{\texttt{Journal}\}$,
$\rho(p_{15}) = (\texttt{title}, \text{"}\texttt{J. DB}\text{"}), \delta(p_{15}) = n_6$,
$\rho(p_{16}) = (\texttt{year}, \text{"}\texttt{2020}\text{"}), \delta(p_{16}) = n_6$,
$\rho(p_{17}) = (\texttt{vol}, \text{"}\texttt{30}\text{"}), \delta(p_{17}) = n_6$,
$\rho(p_{18}) = (\texttt{if}, \text{"}\texttt{4.321}\text{"}), \delta(p_{18}) = n_6$,
$\gamma(p_{18}, \texttt{source}) = \text{"}\texttt{Clarivate}\text{"}$,
$\sigma(e_1) = (n_2, n_1), \lambda(e_1) = \{\texttt{has\_author}\}$,
$\rho(p_{19}) = (\texttt{order}, \text{"}\texttt{1}\text{"}), \delta(p_{18}) = e_1$,
$\sigma(e_2) = (n_2, n_3), \lambda(e_2) = \{\texttt{booktitle}\}$,
$\rho(p_{20}) = (\texttt{pages}, \text{"}\texttt{\{start: 111, end: 121\}}\text{"}), \delta(p_{20}) = e_2$,
$\sigma(e_3) = (n_2, n_4), \lambda(e_3) = \{\texttt{has\_author}\}$,
$\rho(p_{21}) = (\texttt{order}, \text{"}\texttt{2}\text{"}), \delta(p_{21}) = e_3$,
$\sigma(e_4) = (n_5, n_2), \lambda(e_4) = \{\texttt{cites}\}$,
$\sigma(e_5) = (n_5, n_4), \lambda(e_5) = \{\texttt{has\_author}\}$,
$\rho(p_{22}) = (\texttt{order}, \text{"}\texttt{1}\text{"}), \delta(p_{22}) = e_5$,
$\sigma(e_6) = (n_5, n_6), \lambda(e_6) = \{\texttt{published\_in}\}$,
$\rho(p_{23}) = (\texttt{pages}, \text{"}\texttt{\{start: 222, end: 232\}}\text{"}), \delta(p_{23}) = e_6$,

C. PROPERTY GRAPH SCHEMA (FORMAL DEFINITION)

The term *data schema* is related to a way to describe the data structure and enforce its consistency. In this sense, a graph schema allows to define types of nodes, types of edges,

types of properties, and restrictions for these types. Next, we present a formal definition of graph schema.

Recall that **L** is an infinite set of labels and **T** is a finite set of datatypes (e.g., String, Integer, Date, etc.).

Definition 2: A property graph schema, also called a graph type, is a tuple $S = (gt, T_N, T_E, T_P, gtp, \Lambda, \Sigma, \phi, \Delta, \text{Min}, \text{Max}, T_P^{unique}, T_P^{opt}, T_P^{null})$ where:

1) $gt \in \mathbf{L}$ is the label of the graph type;
2) T_N is a finite set of identifiers for *node types*;
3) T_E is a finite set of identifiers for *edge types*;
4) T_P is a finite set of identifiers for *property types*;
5) $gtp : \mathbf{L} \nrightarrow \mathbf{T}$ is a partial function that defines a set of property types that are specific to S;
6) It applies that T_N, T_E and T_P are disjoint sets;
7) $\Lambda : (T_N \cup T_E) \nrightarrow \text{SET}^+(\mathbf{L})$ is a partial function that associates each node/edge type with a non-empty set of labels;
8) $\Sigma : T_E \rightarrow (T_N \times T_N)$ is a total function that associates each edge type with a pair of node types;
9) $\phi : T_P \rightarrow \mathbf{L} \times \mathbf{T}$ is a total function that defines the label and datatype for each property type;
10) $\Delta : T_P \rightarrow (T_N \cup T_E \cup T_P)$ is a total function that associates each property type with a node type, an edge type, or another property type.
11) $\text{Min} : T_p \nrightarrow \mathbb{Z}^{0+}$ is a partial function that allows to specify the minimum cardinality (i.e., a positive integer) for some property types.
12) $\text{Max} : T_p \nrightarrow \mathbb{Z}^{0+}$ is a partial function that allows to specify the maximum cardinality for some property types.
13) $T_P^{unique} \subset T_P$ is the subset of property types that are unique inside a node/edge type;
14) $T_P^{opt} \subset T_P$ is the subset of property types that are optional inside a node/edge type;
15) $T_P^{null} \subset T_P$ is the subset of property types whose value can be NULL.

Next, we describe a property graph schema corresponding to the property graph presented in Figure 2.

$gt = \texttt{Publications}$,
$T_N = \{nt_1, nt_2, nt_3, nt_4, nt_5\}$,
$T_E = \{et_1, et_2, et_3, et_4\}$,
$T_P = \{pt_1, pt_2, pt_3, pt_4, pt_5, pt_6, pt_7, pt_8, pt_9, pt_{10}, pt_{11}, pt_{12}, pt_{13}, pt_{14}, pt_{15}, pt_{16}, pt_{17}, pt_{18}, pt_{19}\}$,
$gtp(\texttt{s}) = (\texttt{about, String})$,
$\Lambda(nt_1) = \{\texttt{Author}\}$, $\Lambda(nt_2) = \{\texttt{Entry, InProceedings}\}$,
$\Lambda(nt_3) = \{\texttt{Entry, Article}\}$, $\Lambda(nt_4) = \{\texttt{Proceedings}\}$,
$\Lambda(nt_5) = \{\texttt{Journal}\}$,
$\Lambda(et_1) = \{\texttt{has\_author}\}$, $\Sigma(et_1) = (nt_2, nt_1)$
$\Lambda(et_2) = \{\texttt{booktitle}\}$, $\Sigma(et_2) = (nt_2, nt_4)$
$\Lambda(et_3) = \{\texttt{has\_author}\}$, $\Sigma(et_3) = (nt_3, nt_1)$
$\Lambda(et_4) = \{\texttt{cites}\}$, $\Sigma(et_4) = (nt_3, nt_2)$
$\Lambda(et_5) = \{\texttt{published\_in}\}$, $\Sigma(et_5) = (nt_3, nt_5)$
$\phi(pt_1) = (\texttt{fname, String})$, $\Delta(pt_1) = nt_1$
$\phi(pt_2) = (\texttt{lname, String})$, $\Delta(pt_2) = nt_1$
$\phi(pt_3) = (\texttt{title, String})$, $\Delta(pt_3) = nt_2$
$\phi(pt_4) = (\texttt{numpages, Integer})$, $\Delta(pt_4) = nt_2$
$\phi(pt_5) = (\texttt{keyword, String})$, $\Delta(pt_5) = nt_2$
$\phi(pt_6) = (\texttt{title, String})$, $\Delta(pt_6) = nt_4$
$\phi(pt_7) = (\texttt{year, Integer})$, $\Delta(pt_7) = nt_4$
$\phi(pt_8) = (\texttt{month, String})$, $\Delta(pt_8) = nt_4$
$\phi(pt_9) = (\texttt{title, String})$, $\Delta(pt_9) = nt_3$
$\phi(pt_{10}) = (\texttt{numpages, Integer})$, $\Delta(pt_{10}) = nt_3$
$\phi(pt_{11}) = (\texttt{keyword, List (String)})$, $\Delta(pt_{11}) = nt_3$
$\phi(pt_{12}) = (\texttt{if, Float})$, $\Delta(pt_{12}) = nt_5$
$\phi(pt_{13}) = (\texttt{title, String})$, $\Delta(pt_{13}) = nt_5$
$\phi(pt_{14}) = (\texttt{year, Integer})$, $\Delta(pt_{14}) = nt_5$
$\phi(pt_{15}) = (\texttt{vol, Integer})$, $\Delta(pt_{15}) = nt_5$
$\phi(pt_{16}) = (\texttt{order, Integer})$, $\Delta(pt_{16}) = et_1$
$\phi(pt_{17}) = (\texttt{pages, Struct})$, $\Delta(pt_{17}) = et_2$
$\phi(pt_{18}) = (\texttt{order, Integer})$, $\Delta(pt_{18}) = et_3$
$\phi(pt_{19}) = (\texttt{pages, Struct})$, $\Delta(pt_{19}) = et_5$,
$T_P^{unique} = \{pt_3, pt_9\}$
$T_P^{opt} = \{pt_{11}\}$
$T_P^{null} = \{pt_8\}$

IV. YARS-PG SYNTAX

In this section, we describe the components of a YARS-PG serialization. The YARS-PG specification is divided in 5 levels: Core (see FR1-FR8 and FR11 in Subsection II-B), Schema (see FR9 in Subsection II-B), Metadata (see FR10 in Subsection II-B), Metaproperties (see FR10 in Subsection II-B), Variables (see FR12 in Subsection II-B), and Graph (see FR10 in Subsection II-B). Tools may provide varying levels of support for YARS-PG, but they must support all elements within the selected level. The Core level is mandatory.

In Listing 1, we present the YARS-PG serialization of the property graph shown in Figure 2. Next, we will use this example to explain the components of YARS-PG.

A. CORE

The Core level includes the declaration of nodes, edges, properties and comments. A one-line comment is declared by using the character #.

A node declaration begins with a node identifier, followed by an optional list of node labels and optional node properties. Examples of node declarations are shown in lines 31 to 36 of Listing 1. The first node has `Author01` as identifier, `Author` as label, `fname` and `lname` as properties with values `"John"` and `"Smith"` respectively. The third node (line 33) is identified by `EI01`, has two labels (`Entry` and `InProceedings`) and three properties (`title`, `numpages` and `keyword`).

An edge declaration begins with a source node identifier, followed by an edge identifier (optional), the labels of the edge, the properties of the edge, and the target node identifier. An edge can be directed (`->`) or undirected (`-`). Examples of edge declarations are shown in lines 39 to 44 of Listing 1. The second edge (line 40) has `EI01` as source node identifier,

```
1  # Metadata
2  +[":foaf": "<http://xmlns.com/foaf/0.1/>"]
3  +["foaf:maker": "Łukasz Szeremeta and Dominik
       Tomaszuk"]
4
5  # Graph schema
6  S/Publications/["about": String]
7
8  # Graph
9  /Publications/["about": "My publications..."]
10
11 # Variables
12 $title_numpages = "title": String UNIQUE, "
       numpages": Integer
13 $start_end = "pages": Struct("start": Integer, "
       end": Integer)
14 $order1 = "order": "1"
15 $journal = "if": "4.321" @<"source": "Clarivate">,
        "title": "J. DB", "year": "2020", "vol": "30"
16
17 # Node schemas
18 S(NS1 {"Author"}["fname": String, "lname": String
       ])+["created": "yesterday"]
19 S(NS2 {"Entry", "InProceedings"}["title": String
       UNIQUE, "numpages": Integer, "keyword": String
        OPTIONAL])
20 S(NS3 {"Entry", "Article"}[$title_numpages, "
       keyword": List(String MIN 1 MAX 5) OPTIONAL])
21 S(NS4 {"Proceedings"}["title": String, "year":
       Integer, "month": String])
22 S(NS5 {"Journal"}["if": Float @<"source": String
       NULL>, "title": String, "year": Integer, "vol
       ": Integer NULL])
23
24 # Edge schemas
25 S(NS2)-({"has_author"}["order": Integer])->(NS1)
26 S(NS2)-({"booktitle"}[$start_end])->(NS4)
27 S(NS3)-({"cities"})->(NS2)
28 S(NS3)-({"published_in"}[$start_end])->(NS5)
29
30 # Nodes
31 (Author01 {"Author"}["fname": "John", "lname": "
       Smith"]) #Author01
32 (Author02 {"Author"}["fname": "Alice", "lname": "
       Brown"])
33 (EI01 {"Entry", "InProceedings"}["title": "
       Serialization for...", "numpages": "10", "
       keyword": "Graph database"])
34 (EA01 {"Entry", "Article"}["title": "Property
       Graph...", "numpages": "10", "keyword": ["
       Query", "Graph"]])
35 (Proc01 {"Proceedings"}["title": "BDAS", "year":
       "2018", "month": "May"])
36 (Jour01 {"Journal"}[$journal])
37
38 # Edges
39 (EI01)-({"has_author"}[$order1])->(Author01)
40 (EI01)-({"has_author"}["order": "2"])->(Author02)
41 (EA01)-({"has_author"}[$order1])->(Author02)
42 (EA01)-({"cites"})->(EI01)
43 (EI01)-({"booktitle"}["pages": {"start": "111", "
       end": "121"}])->(Proc01)
44 (EA01)-({"published_in"}["pages": {"start": "222",
        "end": "232"}])->(Jour01)
```

LISTING 1. Example of YARS-PG serialization.

followed by `has_author` as label, property `order` with value `2`, and `Author02` as a target node identifier.

A property is represented as a pair `k:v` where `k` is the property label and `v` is the property value. A property value could be simple (e.g., a string) or complex (e.g., an array). For example, in Line 34 we can see that the property `numpages` has the simple value `10`, and the property `keyword` has a complex value consisting of a list of strings (“`Query`” and “`Graph`”).

B. SCHEMA

YARS-PG allows to define node types, edge types, property types and graph types. Additionally, YARS-PG supports primitive and complex datatypes. The following primitive datatypes are supported: Bool, String, Bytes, Integer, Unsigned integer, Decimal, Float, DateTime, LocalDateTime, Date, Time, LocalTime, and Duration. User-defined datatypes are also allowed.

The following complex datatypes are supported: Multiset, Set, List, Distinct list, and Struct. A multiset (`MULTISET`) is an unordered collection of elements allowing duplicates. A set (`SET`) is an unordered collection of elements where duplicate elements are not allowed. A list (`LIST`) is an sorted collection of elements allowing duplicates. A distinct list (`DLIST`) is an sorted collection of elements where duplicate elements are not allowed. A structure (`STRUCT`) is an unordered collection of key/value pairs where the value indicates a primitive or a complex datatype.

The declaration of a node type starts with the letter `S`, followed by an identifier, a set of labels (optional), and a set of property types (optional). Examples of node type declarations are shown in lines 18 to 22 of the Listing 1. For instance, the node type defined in line 21 has the identifier `NS4`, the label `Proceedings`, and the property types `title`, `year` and `vol`. A node following this node type declaration is shown in line 35.

An edge type declaration begins with the letter `S`, followed by a source node identifier (mandatory), a set of labels (optional), a set of edge properties (optional), and a target node identifier (mandatory). An edge type can be declared as directed (`->`) or undirected (`-`). Examples of edge type declarations are shown in lines 25 to 28 of Listing 1. For instance, the edge type defined in line 27 has `NS3` as the source node identifier, `cities` as label, and `NS2` as the target node identifier. An edge following this edge type declaration is shown in line 42.

A simple property type declaration has the structure `k:t` where `k` is a property name and `t` is a datatype. The keywords `MIN`, `MAX`, `OPTIONAL`, `UNIQUE` and `NULL` can be used to add restrictions to a property type. The `MIN` and `MAX` keywords allow to specify the minimum and maximum number of elements in a list respectively. The `OPTIONAL` keyword indicates that a given property may or may not occur (similar to `MIN 0 MAX 1`). For example, the line 20 in Listing 1 defines that the property `keyword` is optional and could have a minimum of 1 and a maximum of 5 strings.

Assume that *pt* is a property type defined as part of a node/edge type *t*. If *pt* includes the keyword `UNIQUE` then all the nodes/edges following the type *t* must have a different value for the property type *pt*. For example, the definition of the property type `title` shown in the line 19 of Listing 1,

indicates that all the nodes following the node schema `NS2` must have a distinct value for their property `title`.

The `NULL` keyword defines that the value for a given property could be null. An example of this is shown in the line 22 of Listing 1.

C. METADATA

A metadata declaration begins with the symbol +, followed by a list of properties enclosed by square brackets. The scope of a metadata declaration depends on its location. A metadata declaration located in a single line applies for all the document (e.g. see line 2 in Listing 1). A metadata declaration located at the end of a statement applies just for such statement (e.g. see line 18 in Listing 1).

D. METAPROPERTIES

A metaproperty is a property associated with another property. Hence, a metaproperty allows to include information about a property. A metaproperty declaration begins with the character `@`, followed by a list of properties enclosed by the symbols < and >. Examples of metaproperty declarations are shown in lines 15 and 22 of Listing 1.

E. VARIABLES

In YARS-PG, a variable can be used to store properties. A variable declaration begins with the character `$`, followed by a variable name, the character =, and a list of property types. For example, the line 12 of Listing 1 shows the declaration of the variable `title_numpages`.

For referencing a variable, the user must use the character `$` followed by the variable name. For instance, the variable `title_numpages` is used in the line 20 of Listing 1. Variables can be used more than once.

F. GRAPH TYPES AND GRAPHS

A graph type declaration begins with the string `S/`, followed by a graph identifier, the character `/`, and a list of property declarations (optional). An example of graph type declaration is shown in line 6 of Listing 1.

A graph declaration starts with the character `/`, followed by a graph identifier, the character `/`, a list of graph labels (optional), and a list of properties (optional). An example of graph declaration is shown in the line 9 of Listing 1.

YARS-PG enables assigning nodes and edges to one or more graphs. This can be accomplished by adding the name of the graph (e.g., `/graphname/`) at the end of the node or edge declaration.

The YARS-PG grammar in ANTLR4 [21] and EBNF notation,[1] along with supplementary examples, are available in [22].

[1] https://www.w3.org/TR/REC-xml/#sec-notation

V. FROM PROPERTY GRAPHS TO YARS-PG

This section delineates a systematic approach for mapping various components of a property graph (as described in section III-B) to YARS-PG (as described in section III-C).

A. STEPS FOR MAPPING A GRAPH TYPE

1) Create the schema definition in the syntax using the label of S (g).
2) Include any graph-specific property types (gt) in this definition.

B. STEPS FOR MAPPING A NODE TYPE

1) For each node type $n \in T_N$, create a node schema definition.
2) Use the Λ function to retrieve the labels for each node type and include them in the schema.
3) Add properties to each node type using the ϕ and Δ functions to determine the properties associated with each node type.

C. STEPS FOR MAPPING AN EDGE TYPE

1) For each edge type $e \in T_E$, create an edge schema definition.
2) Use the Λ function to retrieve the labels for each edge type.
3) Use the Σ function to determine the node types connected by the edge type.
4) Add properties to each edge type using the ϕ and Δ functions to determine the properties associated with each edge type.

D. STEPS FOR MAPPING A PROPERTY TYPE

1) For each property $p \in T_P$, determine its label and datatype using the ϕ function.
2) Map the constraints (i.e. Min, Max, UNIQUE and NULL) defined by each property type.

E. STEPS FOR MAPPING A GRAPH

1) Create the graph definition in the syntax using the label of G (g).
2) Include any graph-specific properties (gp) in this definition.

F. STEPS FOR MAPPING A NODE

1) For each node $n \in N$, create a node declaration.
2) Use λ to retrieve the labels for each node.
3) For each property $p \in P$, if $\delta(p)$ refers to a node, include this property in the node's definition, using ρ to get the label-value pair.

G. STEPS FOR MAPPING AN EDGE

1) For each edge $e \in E$, create an edge declaration.
2) Use σ to determine the connected nodes (source and target) for each edge.
3) Use λ to retrieve labels for each edge.
4) Similarly to nodes, include properties for edges where $\delta(p)$ refers to an edge.

H. MAPPING A PROPERTY

For each property $p \in P$, if it is a metaproperty (identified by γ), modify the property's definition in its respective node or edge to include the metaproperty.

I. MAPPING A SUBPROPERTY

For properties that have subproperties (as defined by δ), recursively include these in the syntax, maintaining the hierarchy of properties.

VI. EXTENSIONS OF YARS-PG

This section explores the serialization of RDF data and the canonicalization process within the YARS-PG data format, enhancing its functionality and interoperability in the realm of Semantic Web technologies [23].

A. SERIALIZING RDF

RDF [24] is an abstract graph data model with several serialization formats being essentially specialized file formats. YARS-PG can store RDF triples without changing the grammar and become an RDF serialization format.

The method for transforming an RDF graph into YARS-PG is presented below.

Require: RDF Graph *RG*
  for all t in RG **do**
    subject:= readSubject(t)
    sid := gen()
    if typeOf(subject) == 'IRI' **then**
      createNode(sid, 'IRI', subject)
    else
      createNode(sid, 'BNode', subject)
    end if
    object := readObject(t)
    oid := gen()
    if typeOf(object) == 'Literal' and datatype(object) == 'string' and lang(object) == true **then**
      lang := readLang(object)
      createNode(oid, 'Literal', object, 'xsd:string', lang)
    else if typeOf(object) == 'Literal' and (datatype(object) != 'string' or lang(object) == false) **then**
      datatype := readDatatype(object)
      createNode(oid, 'Literal', object, datatype)
    else if typeOf(object) == 'IRI' **then**
      createNode(oid, 'IRI', object)
    else
      createNode(oid, 'BNode', object)
    end if
    predicate := readPredicate(t)
    createEdge(sid, 'IRI', predicate, oid)
  end for

The algorithm takes an RDF graph (*RG*) as input. An RDF graph consists of triples, each comprising a subject, a predicate, and an object. The algorithm iterates through each triple (`t`) in the RDF graph; extracts the subject of the triple (`subject`) using the `readSubject` function; generates a unique identifier for the subject (`sid`) using the `gen()` function; determines the type of the subject (IRI or Blank Node); and creates a corresponding node using the `createNode` function. Similar steps are for objects, but additionally, literals (with datatypes and with/without languages) are supported. The algorithm, next, extracts the predicate of the triple (predicate) using the `readPredicate` function; creates an edge between the subject and object nodes, associating it with the predicate.

```
1 (s01 {"IRI"}["@value": "http://en.wikipedia.org/
      wiki/Helium"])
2 (o01 {"Literal"}["@value": "2", "@datatype": "http
      ://www.w3.org/2001/XMLSchema#integer"])
3 (s01)-({"IRI"}["@value": "http://example.org/
      elements/atomicNumber"])->(o01)
```

LISTING 2. An RDF triple in YARS-PG.

In Listing 2, we present a simple example of representing RDF in YARS-PG.

B. CANONICALIZATION OF YARS-PG

Canonicalization is the process of transforming data into a standardized and normalized form. This is crucial for computational tasks that require consistent data representation or comparison.

The algorithm for YARS-PG canonicalization includes the following steps:

1) **Declaration roll-Up:** Transform multi-line declarations into singular-line declarations to enhance the document structure's uniformity and improve clarity.
2) **Comment removal:** Remove all comment lines from the YARS-PG document.
3) **Metadata removal:** Remove all metadata from the YARS-PG document.
4) **Variables removal:** Insert variable values where they are used and remove variable declarations.
5) **Declaration reordering:** Reorder declarations systematically based on the following order: graph type declarations, graph declarations, document metadata declarations, node type declarations, edge type declaration, node declarations and edge declarations.
6) **Empty line removal:** Eliminate whitespaces by removing empty lines containing only line feed `LF` or `CR` characters to compress the document and eliminate redundant spacing.
7) **Whitespace removal:** Eradicate all unnecessary white spaces, including spaces (`U+0020`) and tabs (`U+0009`), between serialized elements to reduce file size and eliminate non-semantic discrepancies.

The above algorithm provides a consistent and replicable approach to YARS-PG canonicalization. It ensures that YARS-PG documents with syntactic differences but having the same meaning are serialized uniformly, promoting coherence and comparability across platforms and systems.

```
S/Publications/["about":String]
/Publications/["about":"Mypublications..."]
S(NS1{"Author"}["fname":String,"lname":String])
S(NS2{"Entry","InProceedings"}["title":
    StringUNIQUE,"numpages":Integer,"keyword":
    StringOPTIONAL])
S(NS3{"Entry","Article"}["title":StringUNIQUE,"
    numpages":Integer,"keyword":List(
    StringMIN1MAX5)OPTIONAL])
S(NS4{"Proceedings"}["title":String,"year":Integer
    ,"month":String])
S(NS5{"Journal"}["if":Float@<"source":StringNULL
    >,"title":String,"year":Integer,"vol":
    IntegerNULL])
S(NS2)-({"has_author"}["order":Integer])->(NS1)
S(NS2)-({"booktitle"}["pages":Struct("start":
    Integer,"end":Integer)])->(NS4)
S(NS3)-({"cities"})->(NS2)
S(NS3)-({"published_in"}["pages":Struct("start":
    Integer,"end":Integer)])->(NS5)
(Author01{"Author"}["fname":"John","lname":"Smith
    "])
(Author02{"Author"}["fname":"Alice","lname":"Brown
    "])
(EI01{"Entry","InProceedings"}["title":"
    Serializationfor...","numpages":"10","keyword
    ":"Graphdatabase"])
(EA01{"Entry","Article"}["title":"PropertyGraph
    ...","numpages":"10","keyword":["Query","Graph
    "]])
(Proc01{"Proceedings"}["title":"BDAS","year
    ":"2018","month":"May"])
(Jour01{"Journal"}["if":"4.321"@<"source":"
    Clarivate">,"title":"J.DB","year":"2020","vol
    ":"30"])
(EI01)-({"has_author"}["order":"1"])->(Author01)
(EI01)-({"has_author"}["order":"2"])->(Author02)
(EA01)-({"has_author"}["order":"1"])->(Author02)
(EA01)-({"cites"})->(EI01)
(EI01)-({"booktitle"}["pages":{"start":"111","end
    ":"121"}])->(Proc01)
(EA01)-({"published_in"}["pages":{"start":"222","
    end":"232"}])->(Jour01)
```

LISTING 3. Example in YARS-PG after canonicalization.

Listing 3 shows a running example after canonicalization. The YARS-PG running example before canonicalization can be found in Listing 1.

VII. EVALUATION

This section evaluates the YARS-PG format, demonstrating its parsing efficiency and scalability across various programming environments. The experiments conducted utilized both real-life and synthetic datasets. Furthermore, this section compares YARS-PG with other data formats regarding non-functional and functional requirements.

A. USE OF YARS-PG

In this section we describe the use of YARS-PG in various contexts.

1) SET UP

Our series of experiments were performed on Ubuntu 22.04.3 LTS, featuring an Intel Core i5-11400H processor, 32 GB of single-channel RAM operating at 3200 MHz, and a high-speed 1TB NVMe SSD using PCIe Gen3 × 4.[2] With the Java runtime environment installed at version 17.0.9 and Python 3.10.12, we executed 10 runs to establish average execution times. The YARS-PG parsers in Java and Python can be found in [22], along with ready-to-use packages in the YARS-PG release.[3]

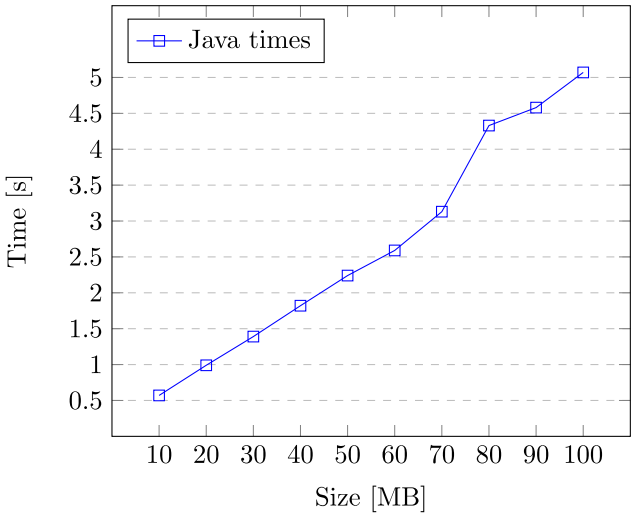

FIGURE 3. Parsing time of real-life data based on YARS-PG grammar in Java.

2) REAL-LIFE DATA

As a real-world dataset, the Mathematical Library Knowledge Graph (MMLKG) [25] is used. It comprises computer-verified mathematical definitions, statements, and proofs from the Mizar proof-checker (Mizar Mathematical Library). We prepared incremental dataset segments in the YARS-PG format: the first 10, 20, 30, 40, 50, 60, 70, 80, 90, and 100 MB, without breaking lines. The files are available at [26].

Both languages demonstrate the relationship between file size and parsing times in comparing file parsing plots between Java and Python. In the Java plot, parsing times increment gradually with file size, suggesting a relatively linear correlation with a gentle slope. On the other hand, the Python plot reveals a steeper increase in parsing times as file size grows, indicating a potentially higher computational load. Java generally exhibits lower parsing times across the provided dataset, implying a more efficient parsing speed compared to Python. The scaling behavior differs between the two languages, with Python showing a more pronounced increase in parsing times. Both plots indicate that parsing times for YARS-PG follow a linear trend, and in both cases, parsing times are satisfactory. The YARS-PG grammar is constructed to be language-independent, showcasing its versatility and adaptability across different programming languages.

[2]Disk read: 2282.85 MB/sec.
[3]https://github.com/lszeremeta/yarspg/releases/tag/v4.0.0

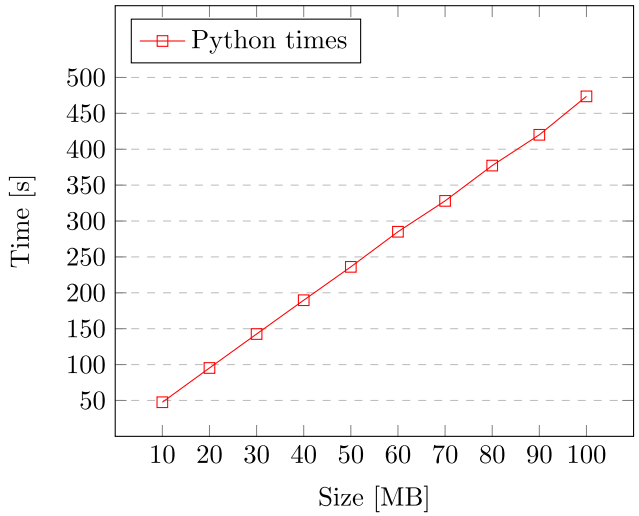

FIGURE 4. Parsing time of real-life data based on YARS-PG grammar in Python.

TABLE 2. Characteristics of generated data.

Nodes	Edges	File size [MB]
10000	16000	1.5
20000	32000	3.0
30000	48000	4.6
40000	64000	6.1
50000	80000	7.6
60000	96000	9.1
70000	112000	11.0
80000	128000	13.0
90000	144000	14.0
100000	160000	16.0

3) BENCHMARK DATA

In addition to the dataset based on real data, we also used data from the Knows 1.0.0 benchmark [27]. This benchmark allows the generation of nodes and edges, supporting multiple output formats, and providing visualization options. The nodes are labeled as `Person` with unique numerical identifiers. Each node features properties for `firstName` and `lastName` with randomly assigned values. The edges have random nodes and avoid cycles. They are labeled as `knows` and include a `createDate` property with a random date.

Knows 1.0.0 is available on PyPI[4] and Docker Hub[5] and can be directly executed from Python's source code. Table 2 shows the characteristics of the generated data. Generated files are available at [28].

Figure 5 and Figure 6 present a comparison between Java and Python across various node counts, ranging from 10,000 to 100,000. It shows a series of values for each language corresponding to each node count. For Java, the values start at 0.23 for 10,000 nodes and incrementally increase to 0.96 at 100,000 nodes. In contrast, Python starts at a value of 9 for 10,000 nodes and progresses to 92.21 for 100,000 nodes. The consistent trend observed is that Java's values are significantly lower than Python's for the same number of nodes, suggesting better performance or efficiency in this specific context. Both plots indicate that parsing times for YARS-PG follow a linear trend, and in both cases, parsing times are satisfactory. This trend is maintained across all node counts, indicating a performance difference between the two languages in favor of Java.

[4]https://pypi.org/project/knows/1.0.0/
[5]https://hub.docker.com/r/lszeremeta/knows

TABLE 3. Comparison of graph data formats considering non-functional features.

Data format / Criterion	Descriptiveness	Well-definedness	Flexibility	Expressiveness
GEXF	1.0	1.0	1.0	0.5
GDF	0.0	0.5	0.5	0.0
GML	0.0	1.0	0.0	0.5
GraphML	0.5	1.0	1.0	0.5
Pajek NET	0.5	0.5	0.0	0.5
GraphViz DOT	0.0	1.0	0.5	1.0
UCINET DL	0.0	0.0	0.0	0.5
Tulip TPL	0.5	0.5	0.0	0.0
Netdraw VNA	0.0	0.5	0.5	0.5
DotML	0.5	1.0	0.5	0.5
S-Dot	0.0	0.5	0.5	0.5
GraphSON TP2	0.0	0.5	1.0	0.5
GraphSON TP3	0.0	0.5	1.0	1.0
DGML	0.5	0.5	1.0	0.5
GXL	0.5	1.0	1.0	1.0
CSV	0.0	0.0	0.0	0.0
JGF	0.0	1.0	1.0	0.5
TGF	0.0	0.0	0.0	0.0
YARS-PG	1.0	1.0	1.0	1.0

B. COMPARISON WITH OTHER DATA FORMATS

In this section, we compare YARS-PG with other data formats examining both non-functional and functional aspects, showing their strengths and weaknesses.

1) COMPARISON OF NON-FUNCTIONAL FEATURES

Table 3 presents a comparison of non-functional features across various graph data formats or notations. Each data format is evaluated on five specific criteria: descriptiveness, well-definedness, flexibility, and expressiveness (see Subsection II-C). The criteria are measured on a scale from 0.0 to 1.0, with 1.0 representing the highest score in a particular category.

YARS-PG excels with perfect scores in all categories, demonstrating its comprehensive capabilities. Contrastingly, GEXF, while performing well in descriptiveness,

TABLE 4. Comparison of graph data formats considering functional features.

	Properties			Labels		Edges				Other				Style
Name	Pairs	Multiple	*Null* value	Multiple	Unique	Directed	Undirected	Multiple	Same label	Schema	Metaproperties	Complex datatypes	Variables	
GEXF	●					●	●	●	●	○		●	○⋆	XML
GDF			○△			●		●	○▲	○				Tabular
GML	●		○△			●	○▽	●						Textual
GraphML	●	○			●▼	●	●	●		●			○⋆	XML
Pajek NET				●		●	●	●						Tabular
GraphViz DOT	●	○	○△	●		●	●		●					Textual
UCINET DL	●			●		●	●	●	●					Textual
Tulip TPL		●				●				○				Textual
Netdraw VNA	●		○△		●	●		●	●					Tabular
DotML				●		●		●	●				○⋆	XML
S-Dot				●		●		●	●					Textual
GraphSON TP2	●					●		●	●	●	●			JSON
GraphSON TP3	●	○		●	●	●		●	●	●	●	●		JSON
DGML	●					●		●					○⋆	XML
GXL	●	○			●▼	●	●	●		○			○⋆	XML
CSV	○	○	○	○	○	○	○	○	○					Tabular
JDF					●	●	●	●	●					JSON
TGF				●										Textual
YARS-PG	●	●	●	●	○	●	●	●	●	●	●	●	●	Textual

△ no grammar
▲ labels supported as properties
▽ only global definition
▼ only in the sense of identifiers
⋆ only via DTD XML entities

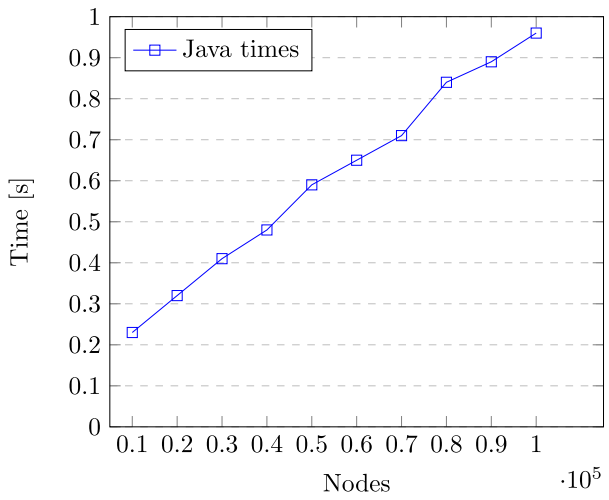

FIGURE 5. Parsing time of benchmark data based on YARS-PG grammar in Java.

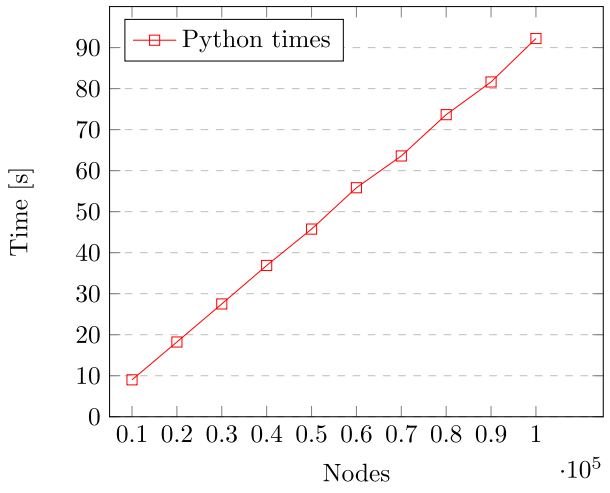

FIGURE 6. Parsing time of benchmark data based on YARS-PG grammar in Python.

well-definedness, and flexibility with scores of 1.0, falls behind in expressiveness with a score of 0.5. Formats like CSV and TGF are at the lower end, scoring 0.0 across all categories, highlighting a significant gap compared to YARS-PG. Other formats, such as GraphML, GraphViz DOT, and GXL, show variability in their capabilities. Formats like GDF, GML, Pajek NET, and others exhibit a mix of low to moderate scores, indicating areas where they are less effective compared to YARS-PG.

2) COMPARISON OF FUNCTIONAL FEATURES

Table 4 compares different serialization formats for property graphs based on functional features. The columns are organized into categories like properties (support for property

pairs, multiple properties, and null values), labels (support for multiple and unique labels), edges (support for directed and undirected edges, multiple edges, and edges with the same label), other features (support for schema, metaproperties, complex datatypes, and variables) and style (XML, JSON, textual, or tabular). Each row represents a different graph serialization format.

YARS-PG stands out as the most feature-rich format in this table, supporting all listed features. It supports property pairs, multiple properties, and null values, which is broader than other formats. It supports multiple labels but not unique labels, a feature less common among the listed formats. YARS-PG supports all edge-related features like directed, undirected, multiple edges, and edges with the same label, which is more comprehensive than other formats like GDF and Tulip TPL. It is marked as unstructured, sharing this characteristic with GEXF and GraphML. YARS-PG falls under the textual format category, similar to GML and GraphViz DOT. GraphML, GraphSON TP3, and GXL support a broad range of features, making them versatile for various use cases. CSV has the least support, with only partial support in all categories, which is understandable given its simple, tabular nature.

VIII. CONCLUSION

A data format plays a pivotal role in the realm of graph data management, serving as the backbone for database exchange, system benchmarking, and data visualization. Within this context, our research introduces YARS-PG, a data format meticulously crafted for serializing property graphs. YARS-PG stands out for its simplicity, extensibility, and platform independence, making it a robust choice for various applications. One of its key strengths lies in its ability to seamlessly accommodate all the features offered by contemporary database systems rooted in the property graph data model. This versatility ensures that YARS-PG is not only user-friendly but also capable of meeting the evolving demands of modern graph-oriented systems, fostering efficient data exchange, benchmarking, and visualization practices.

Future research will concentrate on the binary version of YARS-PG with advanced data compression algorithms. Furthermore, we intend to integrate our serialization with common graph databases. Additionally, we will endeavor to provide tools that facilitate the transformation of data to and from the YARS-PG format.

REFERENCES

[1] *Information Technology—Database Languages SQL—Part 16: Property Graph Queries (SQL/PGQ)*, ISO/IEC Standard 9075-16:2023, Int. Org. Standardization, Geneva, Switzerland, Jun. 2023.
[2] *Information Technology—Database Languages—GQL*, ISO/IEC Standard 39075:2024, International Organization for Standardization, Geneva, Switzerland, 2024.
[3] M. Bastian, S. Heymann, and M. Jacomy, "Gephi: An open source software for exploring and manipulating networks," in *Proc. Int. AAAI Conf. Web Social Media*, Mar. 2009, vol. 3, no. 1, pp. 361–362, doi: 10.1609/icwsm.v3i1.13937.
[4] U. Brandes, M. Eiglsperger, I. Herman, M. Himsolt, and M. S. Marshall, "GraphML progress report structural layer proposal," in *Proc. Int. Symp. Graph Drawing*. Berlin, Germany: Springer, 2001, pp. 501–512, doi: 10.1007/3-540-45848-459.
[5] *The Dot Markup Language*. Accessed: Dec. 4, 2023. [Online]. Available: http://www.Martin-loetzsch.de/DOTML/
[6] *Directed Graph Markup Language (DGML) Reference*. Accessed: Dec. 12, 2023. [Online]. Available: https://docs.microsoft.com/en-us/visualstudio/modeling/directed-graph-markup-language-dgml-reference
[7] R. C. Holt, A. Winter, and A. Schurr, "GXL: Toward a standard exchange format," in *Proc. 7th Work. Conf. Reverse Eng.*, 2000, pp. 162–171, doi: 10.1109/WCRE.2000.891463.
[8] *GraphSON Reader and Writer Library*. Accessed: Dec. 14, 2023. [Online]. Available: https://github.com/tinkerpop/blueprints/wiki/GraphSON-Reader-and-Writer-Library
[9] *TinkerPop Documentation*. Accessed: Dec. 15, 2023. [Online]. Available: http://tinkerpop.apache.org/docs/current/reference/#graphson-reader-writer
[10] A. Bargnesi, A. DiFabio, G. S. William Hayes, C. Benz, H. Pyle, and T. Giggy. Accessed: Dec. 15, 2023. [Online]. Available: http://jsongraphformat.info/
[11] E. Adar, "GUESS: A language and interface for graph exploration," in *Proc. SIGCHI Conf. Human Factors Comput. Syst.*, Apr. 2006, pp. 791–800, doi: 10.1145/1124772.1124889.
[12] V. Batagelj and A. Mrvar, "Pajek—Analysis and visualization of large networks," in *Graph Drawing Software*. Berlin, Germany: Springer, 2004, pp. 77–103, doi: 10.1007/978-3-642-18638-7_4.
[13] B. Cronin, "Getting started in social network analysis with NETDRAW," Univ. Greenwich Bus. School, Faculty Bus., London, U.K., Tech. Rep. Occasional Paper 01/15, 2015. [Online]. Available: https://gala.gre.ac.uk/id/eprint/16200/7/16200%20CRONINGettingStartedin%20SNAwithNETDRAW2015.pdf
[14] Y. Shafranovich, *Common Format and MIME Type for Comma-Separated Values (CSV) Files*, document RFC 4180, Internet Requests for Comments, Oct. 2005. [Online]. Available: http://www.rfc-editor.org/rfc/rfc4180.txt
[15] M. Himsolt. (1997). *GML: A Portable Graph File Format*. [Online]. Available: http://www.uni-passau.de/fileadmin/files/lehrstuhl/brandenburg/projekte/gml/gml-technical-report.pdf
[16] J. Ellson, E. R. Gansner, E. Koutsofios, S. C. North, and G. Woodhull, "Graphviz and dynagraph—Static and dynamic graph drawing tools," in *Graph Drawing Software*. Berlin, Germany: Springer, 2004, pp. 127–148, doi: 10.1007/978-3-642-18638-7_6.
[17] S. P. Borgatti, M. G. Everett, and L. C. Freeman, "UCINET for windows: Software for social network analysis," Analytic Technol., Harvard, MA, USA, Tech. Rep., 2002, vol. 6.
[18] D. Auber, D. Archambault, R. Bourqui, M. Delest, J. Dubois, A. Lambert, P. Mary, M. Mathiaut, G. Melançon, B. Pinaud, B. Renoust, and J. Vallet, "TULIP 5," in *Encyclopedia of Social Network Analysis and Mining*. New York, NY, USA: Springer, 2017, doi: 10.1007/978-1-4614-7163-9315-1.
[19] *S-Dot: A Common Lisp Interface to Graphviz Dot*. Accessed: Jan. 5, 2024. [Online]. Available: http://Martin-loetzsch.de/S-DOT/
[20] M. Roughan and J. Tuke, "The hitchhikers guide to sharing graph data," in *Proc. 3rd Int. Conf. Future Internet Things Cloud*, Aug. 2015, pp. 435–442, doi: 10.1109/FICLOUD.2015.76.
[21] T. Parr, *The Definitive ANTLR 4 Reference*. USA: The Pragmatic Bookshelf, 2013.
[22] Ł. Szeremeta, Feb. 2024, "lszeremeta/yarspg: YARS-PG 4.0.0," *Zenodo*, doi: 10.5281/zenodo.10614187.
[23] T. Berners-Lee, J. Hendler, and O. Lassila, "The semantic web," *Sci. Amer.*, vol. 284, no. 5, pp. 34–43, 2001.
[24] M. Lanthaler, D. Wood, and R. Cyganiak. (Feb. 2014). *RDF 1.1 Concepts and Abstract Syntax*. W3C Recommendation. [Online]. Available: https://www.w3.org/TR/2014/REC-rdf11-concepts-20140225/
[25] D. Tomaszuk, Ł. Szeremeta, and A. Korniłowicz, "MMLKG: Knowledge graph for mathematical definitions, statements and proofs," *Sci. Data*, vol. 10, no. 1, p. 791, Nov. 2023, doi: 10.1038/s41597-023-02681-3.
[26] L. Szeremeta, D. Tomaszuk, and A. Kornilowicz. (Feb. 2024). *Parts of the Mizar Mathematical Library Knowledge Graph (MMLKG) Dataset (YARS-PG Version)*. [Online]. Available: https://figshare.com/articles/dataset/PartsoftheMizarMathematicalLibraryKnowledgeGraphMMLKGdatasetYARS-PGversion/25139015
[27] Ł. Szeremeta, Feb. 2024, "lszeremeta/knows: Knows v1.0.0: First public release!" *Zenodo*, doi: 10.5281/zenodo.10605343.
[28] L. Szeremeta. (Feb. 2024). *Sample Knows Benchmark Outputs in YARS-PG*. [Online]. Available: https://figshare.com/articles/dataset/SampleKnowsbenchmarkoutputsinYARS-PG/25139189

ŁUKASZ SZEREMETA received the bachelor's degree in computer science and M.Sc. degree from the Institute of Computer Science, University of Bialystok, Poland, in 2015 and 2017, respectively. He is currently a Researcher with the Department of Computer Science, University of Bialystok. His current research interests include property graphs, cheminformatics, and semantic web.

DOMINIK TOMASZUK received the M.Sc. degree in computer science from Bialystok University of Technology, Poland, in 2008, and the Ph.D. degree in computer science from Warsaw University of Technology, Poland, in 2014. He is currently a Researcher with the Department of Computer Science, University of Bialystok, Poland. His current research interests include the semantic web, RDF, property graphs, NoSQL databases, and cheminformatics.

RENZO ANGLES received the bachelor's degree in systems engineering from Universidad Católica Santa María, Arequipa, Peru, and the Ph.D. degree in computer science from the Universidad de Chile, in 2009. He is currently an Associate Professor with the Department of Computer Science, Universidad de Talca, Chile. He was a Postdoctoral Researcher at the Department of Computer Science, VU Amsterdam, as part of his participation in the Linked Data Benchmark Council Project. Currently, he participates as a Researcher with the Millennium Institute for Foundational Research on Data, Chile. Specifically, he works in the theory and design of graph query languages, and the interoperability between RDF and graph databases. His research interests include the intersection of graph databases and the semantic web.