Transforming RDF Graphs to Property Graphs using Standardized Schemas

Kashif Rabbani
Aalborg University, Denmark
kashifrabbani@cs.aau.dk

Matteo Lissandrini
University of Verona, Italy
matteo.lissandrini@univr.it

Angela Bonifati
Lyon 1 University, CNRS & IUF, France
angela.bonifati@univ-lyon1.fr

Katja Hose
TU Wien, Austria
katja.hose@tuwien.ac.at

ABSTRACT

Knowledge Graphs can be encoded using different data models. They are especially abundant using RDF and recently also as property graphs. While knowledge graphs in RDF adhere to the subject-predicate-object structure, property graphs utilize multi-labeled nodes and edges, featuring properties as key/value pairs. Both models are employed in various contexts, thus applications often require transforming data from one model to another. To enhance the interoperability of the two models, we present a novel technique, S3PG, to convert RDF knowledge graphs into property graphs exploiting two popular standards to express schema constraints, i.e., SHACL for RDF and PG-Schema for property graphs. S3PG is the first approach capable of transforming large knowledge graphs to property graphs while fully preserving information and semantics. We have evaluated S3PG on real-world large-scale graphs, showing that, while existing methods exhibit lossy transformations (causing a loss of up to 70% of query answers), S3PG consistently achieves 100% accuracy. Moreover, when considering evolving graphs, S3PG exhibits fully monotonic behavior and requires only a fraction of the time to incorporate changes compared to existing methods.

ACM Reference Format:
Kashif Rabbani, Matteo Lissandrini, Angela Bonifati, and Katja Hose. 2025. Transforming RDF Graphs to Property Graphs using Standardized Schemas. In *Companion of the 2025 International Conference on Management of Data (SIGMOD-Companion '25), June 22–27, 2025, Berlin, Germany.* ACM, New York, NY, USA, 14 pages.

1 INTRODUCTION

Knowledge Graphs (KGs) play a pivotal role in various domains by enabling storage and querying information as graph data [20, 29, 36]. KGs are usually modeled either as property graphs (PG [16]) or using the Resource Description Framework (RDF [14]). KGs modeled as PGs are mostly used in social networks, recommender systems, and fraud detection [10], while KGs modeled as RDF are often used in question answering, semantic search, and reasoning [7]. Both types of models rely on basic graph concepts like nodes and edges, but have different modeling approaches, expressivity, serializations, and query paradigms. When modeling KGs in RDF, data is stored in the form of <*subject-predicate-object*> triples. On the other hand, the PG model assigns multiple labels to both nodes and edges. These nodes and edges are further annotated with properties encoded as sets of key-value pairs. Despite the growing prominence of KGs, the interoperability between these two models remains unresolved [4] and therefore poses a significant challenge for users of graph data [22]. The ability to seamlessly translate data modeled as RDF into the PG data model holds profound implications for enhancing data exchange, data integration [23], query performance, and analytics across various domains [4]. Difficulties in transitioning between these models to address evolving business is further causing undesired operational costs [22].

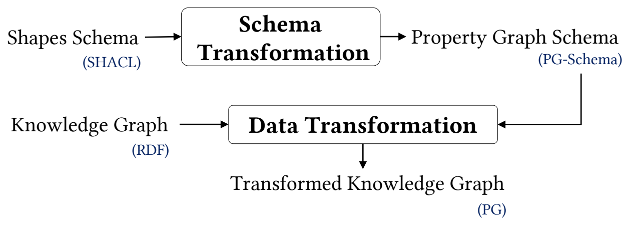


Figure 1: Overview of our proposed S3PG translation

To this end, few approaches [5, 12, 19] have been proposed to convert KGs encoded as RDF triples into PGs with significant shortcomings in mapping or translating between the two models [16]. There are two major limitations in existing transformation techniques [5, 28], i.e., *losslessness* and *monotonicity*. In this work, we refer specifically to schema and data losslessness. We refer to loss of data when converting instance data (i.e., from the ABox) and loss of schema in terms of integrity constraints (e.g., type, uniqueness, or cardinality constraints)[1]. Existing approaches lack these kinds of *losslessness* due to naive conversion methods, for example when mapping multi-valued properties of mixed types, such as simple text values as literals and links in the form of IRIs. These limitations lead to significant data loss during the transformation process. In DBpedia [6] for example, nodes representing music albums contain properties like dbp:writer and dbp:producer. These properties describe attributes of the albums, such as the writer(s) and producer(s). In some cases, these attributes are represented as simple text values (literals), like the name of the writer or producer, while in other

[1]Since PGs do not support a standard language to express ontologies as RDF does, we leave it to future work to define a method to preserve semantics encoded in ontological constraints, e.g., in OWL.

cases, they may be represented as links (IRIs) to other nodes. For instance, in the case of the music album dbr:California_Sunrise, the writers are referenced using IRIs, such as dbr:Billy_Montana, as well as through string literals like 'Tofer Brown'. Existing conversion techniques often assume that these properties consistently lead to homogeneous types of information. For example, they might expect that the dbp:writer property always connects an album to a specific node representing the writer. However, this assumption can break down when dealing with albums where the writer's information is provided only as text, without a specific node. Consequently, this discrepancy results in a considerable loss of information, necessitating extensive manual data preprocessing to ensure a fully data preserving transformation. Additionally, when transforming evolving KGs, existing transformation techniques [5, 28] lack *monotonicity*. This means that any update to the source KG in RDF requires a complete re-computation of the entire data conversion.

To overcome these limitations, we propose the Standardized SHACL Shapes-based PG Transformation (S3PG). Our approach leverages standardized schemas for both RDF and PG data models to ensure the preservation of graph data and integrity constraints in the transformation process, while also preserving monotonicity. In the context of S3PG, the term KG refers either to an RDF graph, which also includes schema information, or to a property graph with its schema. For the RDF data model we consider schema information encoded with SHACL shapes [21], a W3C standard, and for the PG data model we adopt PG-Schema [1], the most recent standard capable of enforcing constraints on property graphs. By leveraging these standardized schemas, we propose the first reliable transformation approach. S3PG involves two distinct transformations: schema transformation and data transformation. Schema transformation converts the SHACL shape schema to PG-Schema, while data transformation converts the instance data from RDF to PG, as depicted in Figure 1. Schema transformation requires the preservation of the semantics encoded by the schema in terms of integrity constraints during the transformation process [37]. Thus, the term *loss* refers to the loss of both instance data, in terms of triples connecting nodes and literals, as well as loss of *integrity constraints* in S3PG. S3PG thus provides the ability to reconstruct the original RDF instance data from the transformed schema and PG data and it preserves the semantics of the integrity constraints encoded by the SHACL shape schema in the transformed schema. Additionally, S3PG ensures monotonicity, ensuring that the PG model does not need full recomputation after updates to the source RDF data. In the example of music albums in DBpedia, SHACL shapes can effectively encode heterogeneous types for writers by specifying types associated with the dbp:writer property. Such constraints can be captured in PG-Schema. Therefore, when a SHACL schema undergoes transformation to PG-Schema using S3PG, these constraints are seamlessly translated, thereby ensuring a transformation that accurately corresponds to the schemas, preserving information and semantics.

We evaluated S3PG using real-world datasets: DBpedia [6] and Bio2RDF Clinical Trials [8]. Our analysis shows that S3PG consistently achieves 100% accuracy, surpassing existing approaches with accuracy rates ranging from 30-99%. Furthermore, S3PG reduces transformation time by one order of magnitude as compared to existing methods. Moreover, S3PG requires only a fraction of the time to incorporate changes in the input RDF KG to the output PG, unlike existing approaches that require the re-computation of the entire transformation.

In this paper, we make the following contributions:

- we present, S3PG, a novel technique to transform RDF knowledge graphs into property graphs using the latest standards, i.e., SHACL [21] and PG-Schema [1];
- we show that S3PG fully preserves information and integrity constraints by efficiently mapping also heterogeneous multi-type multi-value properties;
- SP3G further supports incremental updates in a monotonic manner without the need to re-compute the entire transformation.

This paper is structured as follows. First in Section 2, we explain the preliminaries, formalizing RDF and PG along with their respective schemas. Next, Section 3 formalizes the transformation challenges and properties. Following that, Section 4 presents the S3PG approach for schema and data transformation. The experimental evaluation is presented in Section 5, while Section 6 discusses related work. Finally, Section 7 concludes the paper, providing insights into future research directions.

2 GRAPH SCHEMAS AND DATA MODELS

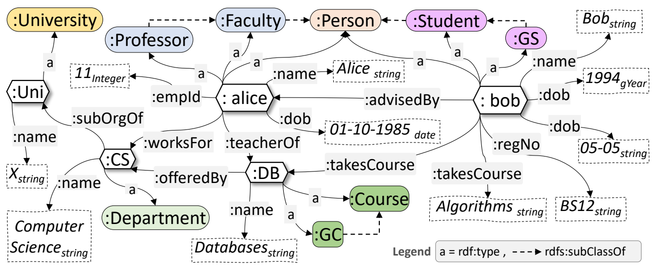
In this section, we present the formal definition of Knowledge Graphs (KGs), along with the basics of how to encode instance data with RDF and PG and how to encode integrity constraints in SHACL and PG-Schema. While it is generally possible to directly translate data modeled in RDF to PGs, it is not always straightforward to establish whether the opposite is true. This is because the PG data model support features that do not have a direct mapping in RDF [41]. Therefore, in S3PG, our focus is on transforming RDF data (the ABox) and the corresponding validating shapes to an equivalent PG and PG-schema, while the conversion from PG to RDF requires a separate study.

2.1 RDF Graphs and Shape Schemas

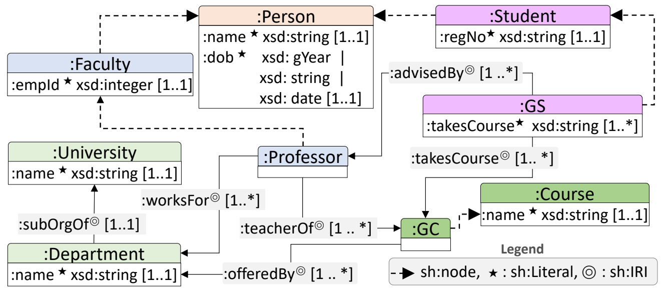
The standard model for encoding KGs is the Resource Description Framework (RDF [14]), which describes data as a set of $\langle s, p, o\rangle$ triples stating that a subject s is in a relationship with an object o through predicate p. We define an RDF graph as follows:

Definition 2.1 (RDF graph). Given pairwise disjoint sets of IRIs $\mathcal{I}$, blank nodes $\mathcal{B}$, and literals $\mathcal{L}$, an RDF Graph $G{:}\langle N, E\rangle$ is a graph with a finite set of nodes $N{\subset}(\mathcal{I}{\cup}\mathcal{B}{\cup}\mathcal{L})$ and a finite set of edges $E{\subset}\{\langle s, p, o\rangle{\in}(\mathcal{I}{\cup}\mathcal{B}) \times \mathcal{I} \times (\mathcal{I}{\cup}\mathcal{B}{\cup}\mathcal{L})\}$.

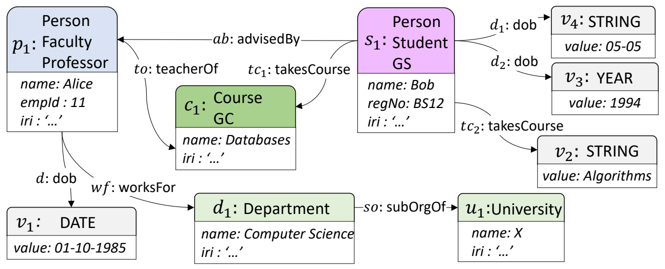
IRIs $\mathcal{I}$ are global identifiers for entities while blank nodes $\mathcal{B}$ are nodes without any defined source identifier and $\mathcal{L}$ represents the set of literals. We distinguish two subsets of IRIs $\mathcal{I}$: predicates $\mathcal{P}$ and classes $\mathcal{C}$. The set of predicates $\mathcal{P}{\subset}\mathcal{I}$ is the subset of IRIs that appears in the predicate position p in any $\langle s, p, o\rangle{\in}G$. Here, we include schema elements from RDFS [15], such as `rdfs:Class`, `rdfs:subClassOf`, and `rdfs:Literal`, that are needed for the interpretation of the shape schema defined below. Thus, among the predicates $\mathcal{P}$, we identify the type predicate `a`$\in\mathcal{P}$ (which is an abbreviation for rdf:type [43] or wdt:P31 in WikiData [40]) and `rdfs:subClassOf`$\in \mathcal{P}$. The former connects all entities that are instances of a class to the node representing the class itself, i.e.,



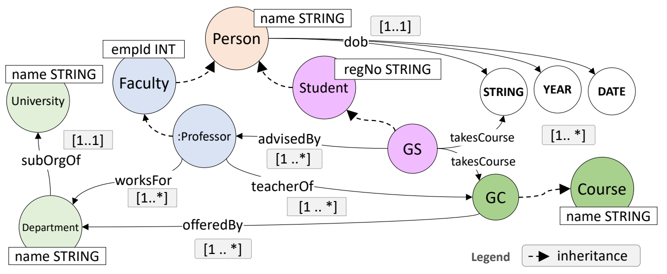
(a) RDF graph for a Graduate Student (GS) taking Graduate Course (GC) advised by a Professor in a particular Department of a University.


:Person
:name ⋆ xsd:string [1..1]
:dob ⋆ xsd: gYear | xsd: string | xsd: date [1..1]
:Student
:regNo⋆ xsd:string [1..1]
:Faculty
:empId ⋆ xsd:integer [1..1]
:GS
:takesCourse⋆ xsd:string [1..*]
:University
:name ⋆ xsd:string [1..1]
:Professor
:Department
:name ⋆ xsd:string [1..1]
:Course
:name ⋆ xsd:string [1..1]
:GC
Legend sh:node, ⋆ : sh:Literal, ◎ : sh:IRI

(b) SHACL schema for the RDF graph shown in Figure 2a. Cardinality constraints: [minCount, maxCount], with * for undefined cardinality.



(c) Property Graph created by transforming the RDF graph shown in Figure 2a using S3PG.



(d) PG-Schema constructed by transforming the SHACL schema shown in Figure 2b using S3PG.

Figure 2: An example of transforming an RDF graph into a property graph using SHACL and PG-Schema

their type. The latter indicates subclass relationships between elements in $\mathcal{C}$. Thus, $\mathcal{C}$:$\{c{\in}\mathcal{I}\,|\,\exists s{\in}\mathcal{I}$ s.t. $\langle s,$a$|$`rdfs:subClassOf`$, c\rangle{\in}G\}$. Figure 2a shows an example that models the data of a subset of a university database.

Apart from instance data in RDF graphs, S3PG also supports converting its schema given in the form of validating shapes. A shape schema $\mathcal{S}_G$ represents integrity constraints on graph G in the form of node and property shapes. Without loss of generality we define $\mathcal{S}_G$ adopting the syntax used for defining the standard SHACL *core constraint components* [35, 42].

Definition 2.2 (Shape Schema). A shape schema $\mathcal{S}_G$ consists of a set of node shapes $\langle s, \tau_s, \Phi_s\rangle{\in}\mathcal{S}_G$, where s is the shape *name*, $\tau_s{\in}\,\mathcal{C}{\cup}\{s'|\langle s', \tau_{s'}, \phi_{s'}\rangle{\in}\mathcal{S}_G\}$ is the *target class* or a node shape $s'{\in}\mathcal{S}_G$, and Φ_s is a set of property shapes of the form $\phi_s{:}\langle\tau_\text{p}, \text{T}_\text{p}, \text{C}_\text{p}\rangle$, where $\tau_\text{p}{\in}\mathcal{P}$ is called the target property, $\text{T}_\text{p}{\subset}\mathcal{I}$ contains either an IRI defining a *literal type*, e.g., xsd:string, or a set of IRIs – called *class type constraint*, and C_p is a pair $(n,m) \in \mathbb{N} \times (\mathbb{N}{\cup}\{\infty\})$. $n{\le}m$ – called min and max *cardinality constraints*.

Therefore, given a node shape $s{\in}\mathcal{S}_G$ for a *target class* $\tau_s{\in}\mathcal{C}$, Φ_s defines which properties each instance of τ_s can or should be associated with. Furthermore, τ_s can also correspond to one of the node shape names in $\mathcal{S}_G$, meaning that it inherits and extends the constraints for that shape. For instance, the Student shape in Figure 2b, contains a node shape for the target class :Student and enforces a property shape ϕ with property τ_p= :regNo (registration number), which is a literal type constraint T_p= xsd:string, with the cardinality constraints C_p=(1, 1). Similarly, the Graduate Student (GS) node shape contains a property shape ϕ having τ_p=:takesCourse with a class type constraint T_p= :Course, and the cardinality C_p=(1,∞). Further, the GS node shape inherits τ_p= :regNo from Student.

When validating a graph G against a shape schema $\mathcal{S}_G$ with node shape $\langle s, \tau_s, \Phi_s\rangle{\in}\mathcal{S}_G$, we verify that each entity $e{\in}G$ that is an instance of τ_s satisfies all the constraints Φ_s. Note that we use the terms entity and node interchangeably throughout the paper. Thus, we define the semantics of $\mathcal{S}_G$ as follows:

Definition 2.3 (Shape Semantics). Given a node shape $\langle s, \tau_s, \Phi_s\rangle{\in}\mathcal{S}_G$, an RDF graph G, and an entity e s.t. $\langle e, \mathbf{a}, \tau_s\rangle{\in}G$, then s validates e, and we write e conforms to ϕ in G (i.e., $e{\models}_G\phi$), if for every property shape $\phi_s{:}\langle\tau_\text{p}, \text{T}_\text{p}, \text{C}_\text{p}\rangle{\in}\Phi_s$ the following conditions hold:

- If T_p is a literal value type constraint, then for every triple $\langle e, \tau_p, l\rangle \in G$, l is a literal of type T_p.
- If T_p is a set of class value type constraints $\text{T}_\text{p}{=}\{t_1, t_2, ...t_n\}$, then for every triple $\langle e, \tau_p, o\rangle{\in}G$, it holds that $\forall t{\in}\text{T}_\text{p}$, o is an instance of t (or of a subclass of t) and if $\exists S_t{\in}\mathcal{S}_G$, $o{\models}_G S_t$.
- If T_p is a set of node type value-based constraints $\text{T}_\text{p}{=}\{s_1, s_2, ...s_n\}$ where $s_i{\in}\mathcal{S}_G$, then for every triple $\langle e, \tau_p, o\rangle{\in}G$, it holds that $\forall s{\in}\text{T}_\text{p}$, $o{\models}s_i$.
- $n \le |\{(s,p,o) \in G : s = e \land p = \tau_p\}| \le m$, where $\text{C}_\text{p}{=}(n,m)$.

Therefore, if a property shape specifies a literal value type constraint (e.g., enforcing values of type string, or integer), then for every triple $\langle e, \tau_p, l\rangle$ in the RDF graph, object l must be a literal of the specified type T_p. If a property shape specifies a set of class value type constraints in T_p (e.g., lists specific RDF classes), then for every triple $\langle e, \tau_p, o\rangle$ in the RDF graph, object o must be an instance of one of the classes in $\mathsf{T_p}$. Additionally, if any of these classes have associated shapes in the schema, object o must conform to the corresponding shapes. If a property shape specifies a node type constraint in $\mathsf{T_p}$, that is, if it refers directly to a node shape s_i in $\mathcal{S}_G$, then for every triple $\langle e, \tau_p, o\rangle$ entity o must conform to s_i defined by the value of $\mathsf{T_p}$. Further, the number of triples in the RDF graph where e is the subject and τ_p is the predicate must not occur fewer than n times or more than m times with entity e.

When using S3PG, it is essential to have a shape schema $\mathcal{S}_G$ of RDF graph G. Some open RDF graphs [38] are already available with shape schemas. For RDF graphs where SHACL shapes are not readily available, various techniques [9, 13, 17, 25, 30, 31, 33] are available to extract a shape schema either from ontologies or directly from the graph data. This flexibility allows us to capture the implicit schema and constraints in G, ensuring that S3PG can accommodate a wide range of complex RDF data models and maintain data integrity throughout the conversion process.

2.2 Property Graph and PG-Schema

A property graph is a node- and edge- labelled directed attributed multigraph [1]. We assume a countable set of labels $\mathcal{L}$, property names (keys) $\mathcal{K}$, property values $\mathcal{V}$, and records $\mathcal{R}$, where a *record* is a finite-domain partial function $o{:}\mathcal{K}{\rightarrow}\mathcal{V}$ mapping keys from $\mathcal{K}$ to values from $\mathcal{V}$, and we adopt the following definition:

Definition 2.4 (Property Graph [1]). A property graph is defined as a tuple $PG = (N, E, \rho, \lambda, \pi)$ where :

- N is a finite set of nodes;
- E is a finite set of edges such that $N \cap E = \emptyset$;
- $\rho : E \rightarrow (N \times N)$ is a total function mapping edges to ordered pairs of nodes;
- $\lambda : (N \cup E) \rightarrow 2^{\mathcal{L}}$ is a total function mapping nodes and edges to finite sets of labels (including the empty set);
- $\pi{:}(N{\cup}E){\rightarrow}\mathcal{R}$ is a function mapping nodes and edges to records.

We show an example PG in Figure 2c, associating an identifier with every node and edge and employing consistent typographic conventions. For example, d_1 is the identifier for the node representing the *Computer Science* department, Department is its only label, and name and iri are its properties. This PG is a translation of the RDF graph shown in Figure 2a. Therefore, it contains iri as one of the properties in the nodes.

PG-Schema [1] is a recent standardized proposal for schema definition of property graphs. PG-Schema consists of two main components: PG-Types and PG-Keys [2]. The former defines node types based on allowed label and content combinations, edge types based on allowed label, content, and endpoint type combinations, and graph types based on the types of nodes and edges present in the graph. The latter focuses on enforcing constraints on the typed data, which includes integrity constraints, such as keys, participation, and cardinality constraints. PG-Schema supports inheritance between PG-Types to define hierarchies and allows intersections and unions for content types. Content types are the data types that schemas can support for nodes and edges [1]. Hence, content types describe attributes assigned to PG-Types, e.g., a node property of type 'name' can be defined using the content type *'string'*, while an edge might include the property *'since'* taking values of type *'date'*. It also supports defining abstract node types. PG-Types can be open or closed, allowing or restricting the addition of new subtypes. For the formal definition, we follow the original definitiong of PG-Schema mainly focusing on PG-Types and later defining PG-Keys [1]. We define a *formal base type* as pair (L, R), where $L{\subseteq}\mathcal{L}$ and $R{\subseteq}\mathcal{R}$ and $\mathcal{T}$ represent the set of all formal base types.

Definition 2.5 (PG-Schema). A PG schema $\mathcal{S}_{PG} = (N_S, E_S, \nu_S, \eta_S, \gamma_S, \mathcal{K}_S)$ consists of the following components:

- N_S: A disjoint set of node type names;
- E_S: A disjoint set of edge type names;
- $\nu_S : N_S \rightarrow 2^{\mathcal{T}}$: A function mapping node type names to sets of formal base types $\mathcal{T}$;
- $\eta_S : E_S \rightarrow 2^{\mathcal{T}\times\mathcal{T}\times\mathcal{T}}$: A function mapping edge type names to tuples of source node type, target node type, and edge type;
- $\gamma_S : N_S \rightarrow 2^{N_S}$: A function mapping node type names to sets of other node types to form type hierarchies;
- $\mathcal{K}_S$: A set of PG-Keys expressions defining constraints.

For brevity, we refer to the elements of N_S and E_S as node and edge types rather than node and edge type names. Here, ν_S specifies the properties or attributes that nodes of each type can have; η_S defines the allowed connections between nodes; γ_S is used to define hierarchies between node types; and, $\mathcal{K}_S$ includes PG-Keys expressions of the form |FOR| $p(x)$ <qualifier> $q(x, \bar{y})$, where a <qualifier> specifies the kind of constraint (such as uniqueness, participation, and cardinality) while $p(x)$ and $q(x, \bar{y})$ are queries.

We present an example PG-Schema in Figure 2d. Each circled node represents a node type with properties as key-value pairs in a rectangular box. For instance, Person is a node type that must have a single, required property called name with a data type of String. Graduate Student (GS) is a node type that inherits the key-value property regNo from the Student node type and has an edge type property takesCourse, which can be of type String or Graduate Course (GC). Inheritance between node types is represented using dotted arrows, for example Graduate Student (GS) is derived from Student, which in turn is derived from Person, thus forming a hierarchy of node types. The cardinality of an edge is represented as a pair [min, max] on top of it, e.g., a node type GS must have at least one takesCourse edge represented as [1..*]. Additionally, PG-Schema introduces two graph-type options: STRICT and LOOSE, providing different levels of adherence to the defined schema. We use slightly different but connected notions, called *conformance* and *typings*, to deal with strict and loose graph-type options as follows:

Definition 2.6 (PG-Schema Semantics). Given property graph $PG{=}(N, E, \rho, \lambda, \pi)$ and its PG-Schema $\mathcal{S}_{PG} = (N_S, E_S, \nu_S, \eta_S, \gamma_S)$, a node $n{\in}N$ *conforms* to a node type $\tau{\in}N_S$, and we write $n{\models}\tau$, if it conforms to a formal base type in $\upsilon_S(\tau)$. Further, an edge $e{\in}E$ *conforms* to an edge type $\sigma{\in}E_S$, and we write $e{\models}\sigma$, if for the pair $(v_1, v_2){=}\rho(e)$ there is a triple $\langle t_1, t, t_2\rangle{\in}\eta_S(\sigma)$ such that v_1 conforms to t_1, e conforms to t, and v_2 conforms to t_2. Finally, a property graph conforms to its schema ($PG \models \mathcal{S}_{PG}$) if every element in PG conforms to at least one type in $\mathcal{S}_{PG}$.

Following the semantics of PG-Schema [1], conformance indicates the extent to which the data in PG is compliant with the rules defined in the form of types and keys. The typing of PG w.r.t. $\mathcal{S}_{PG}$ is the mapping T: $N{\cup}E \rightarrow 2^{N_S} \cup 2^{N_S}$ for all $v{\in}N$ and $e{\in}E$ defined as $T(v)=\{\tau\in N_S|v\models\tau\}$, $T(e)=\{\tau\in E_S|e\models\tau\}$. Hence $PG \models \mathcal{S}_{PG}$ if T only maps to non-empty sets.

3 TRANSFORMATION CHALLENGES AND PROPERTIES

We divide the problem of transforming an RDF Knowledge Graph G to the corresponding property graph PG into two sub-problems, i.e., the problems of *schema transformation* and *data transformation.* We formally define these problems as follows:

Problem 1 (Schema Transformation). *Given the shape schema $\mathcal{S}_G$ of an RDF graph G, the problem of transforming $\mathcal{S}_G$ to a corresponding PG-Schema $\mathcal{S}_{PG}$ requires to generate the pair $(\mathcal{S}_{PG}, F_{st})$, where $F_{st} : \mathcal{S}_G\rightarrow\mathcal{S}_{PG}$ is a schema transformation function that* maps *node and property shape constraints of $\mathcal{S}_G$ to PG-Types and PG-Keys in $\mathcal{S}_{PG}$ such that $\mathcal{S}_{PG}$ encodes all and only the constraints of $\mathcal{S}_G$.*

Problem 2 (Data Transformation). *Given G, $\mathcal{S}_G$, $\mathcal{S}_{PG}$, and F_{st}, the data transformation from G to PG is the problem of generating the pair (PG, F_{dt}), where $F_{dt}[F_{st}]$:$G\rightarrow PG$ is a function mapping all nodes $n\in N$ and edges $e\in E$ in G into elements of a property graph PG using the mapping between $\mathcal{S}_G$ and $\mathcal{S}_{PG}$ provided by F_{st}, such that the resulting PG conforms to $\mathcal{S}_{PG}$, i.e., $PG\models\mathcal{S}_{PG}$.*

3.1 Transformation Properties

The literature identifies a set of desirable properties for a graph data transformation [37], namely: information preservation, query preservation, semantic preservation, and monotonicity.

Information preservation ensures that there exists a way to recover the original information in G and its schema definition from the graph produced by the translation process.

Definition 3.1 (Information preservation). F_{dt} is an information preserving data transformation function, if there exist computable mappings $\mathcal{M}$:$PG\rightarrow G$ from the transformed property graph PG to the original RDF graph G, and $\mathcal{N}$:$\mathcal{S}_{PG}\rightarrow\mathcal{S}_G$ from the transformed PG-Schema to $\mathcal{S}_G$.

Note that mappings $\mathcal{M}$ and $\mathcal{N}$ are computable if and only if there exist algorithms that, given G and $\mathcal{S}_G$, compute $\mathcal{M}(G)$ and $\mathcal{N}(\mathcal{S}_G)$.

The transformation is said to be *query preserving*, if it guarantees that any query over the source graph can be evaluated over the transformed graph and produces the same result [37]. We consider Cypher [18] as the target query language, and, following the literature [37], given G and a SPARQL query Q over G having its results μ, we define a function $tr(\mu)$ to convert the values of variables returned in the SELECT clause of the SPARQL query Q to equivalent values supported by a Cypher query, s.t. IRIs and blank node ID values are converted to equivalent string representations. Thus, we formally define query preservation as follows:

Definition 3.2 (Query preservation). Given shape schema $\mathcal{S}_G$ and PG-schema $\mathcal{S}_{PG}$=$F_{st}(\mathcal{S}_G)$, the schema transformation function F_{st} and the data transformation function F_{dt} are query preserving if for every query Q over any RDF graph instance $G'\models\mathcal{S}_G$, there exists a query Q^* such that for the resulting PG instance PG'=$F_{dt}(G)$ it holds that $tr(\llbracket Q\rrbracket_{G'})$=$\llbracket Q^*\rrbracket_{PG'}$, where $\llbracket Q\rrbracket_{G'}$ denotes the result of evaluating the SPARQL query Q on G' and $\llbracket Q^*\rrbracket_{PG'}$ denotes the result of evaluating the Cypher query Q^* on PG'.

Further, when considering the constraints defined by the shape schema over the source RDF graph, we require a set of equivalent constraints expressed as PG-schema to be enforced also on the transformed target property graph. In this context, and according to the literature [37], such transformation is called *semantics preserving*. Thus, this work focuses on the semantics of the constraints that dictate the "schema" of the graph and we do not consider more expressive ontological definitions. We define it as follows:

Definition 3.3 (Semantics preservation). F_{dt} is a semantics preserving data transformation function if given a $\mathcal{S}_G$, any G that satisfies $\mathcal{S}_G$ is translated to a *PG* satisfying $\mathcal{S}_{PG}$, and any instance of G not satisfying $\mathcal{S}_G$ is translated to a *PG* not satisfying $\mathcal{S}_{PG}$, i.e., if $G \models \mathcal{S}_G$, then $F_{dt}(G) \models \mathcal{S}_{PG}$, and if $G \not\models \mathcal{S}_G$, then $F_{dt}(G) \not\models \mathcal{S}_{PG}$.

Finally, a transformation is *monotonic* if – when new data is added or deleted – only the corresponding data in the target graph is affected, i.e., there is no need for re-computing the entire transformation [37]. To this end, we extend a previous definition [37] as follows:

Definition 3.4 (Monotonicity). A data transformation function F_{dt} is monotonic if, for any given pair of instances $S_1\models\mathcal{S}_G$ and $S_2\models\mathcal{S}_G$, with $S_1\subseteq S_2$, given $S_\Delta = S_2\backslash S_1$, it holds that $F_{dt}(S_1)\subseteq F_{dt}(S_2)$ and $F_{dt}(S_2)\simeq F_{dt}(S_1)\uplus F_{dt}(S_\Delta)$.

4 STANDARDIZED SHACL SHAPES-BASED PG TRANSFORMATION

We propose S3PG to transform an RDF graph G into a property graph PG using shape schema $\mathcal{S}_G$ and PG-Schema $\mathcal{S}_{PG}$. We first provide schema transformation rules and then data transformation rules that exploit the transformed schema. We will use the example graphs shown in Figure 2 to explain each step of the transformation.

4.1 Schema Transformation

We summarize the alternative types of constraints a graph shape can enforce (as described in the SHACL *core constraint components* [42] and Definition 2.2) in the taxonomy depicted in Figure 3. A node shape can have a node kind, which can either be a single type or multiple types. Each property shape requires the definition of a property or property path and can have cardinality constraints (either a minimum or maximum count) and a node kind. For every shape defining a node kind, the choice is between enforcing a *Single type* constraint or allowing *Multiple Types.*

Our taxonomy is exhaustive of all core options required to enforce typical constraints of practical interest. The conversion of a larger set of constraints expressible in SHACL is outside the scope of this work, especially since, as we will see later, already expressing this subset presents non-trivial challenges that existing techniques are not able to overcome and also pushes the boundaries of the set of constraints that the current PG-Schema definition can express. Therefore, for each entry in the taxonomy, we define how that type of constraint is transformed into PG-Schema with PG-Types and PG-Keys (as a solution to Problem 1). To explain each transformation, we will use syntactical examples from the SHACL schema shown in Figure 2b and the PG-Schema shown in Figure 2d.

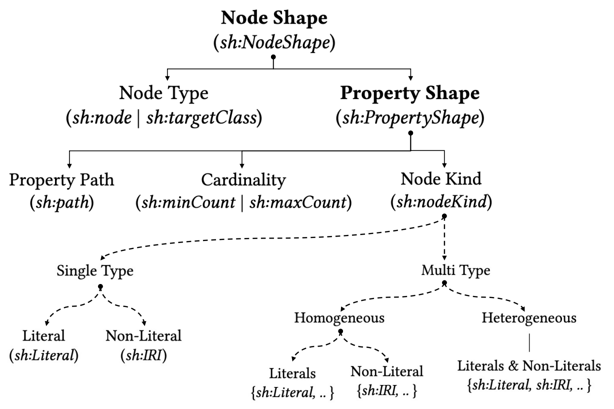


Figure 3: Taxonomy of Node Shape constraints

Node Shape. A node shape defines its type using sh:targetClass or sh:node attributes and associates properties using sh:property. Figure 4 (a,b) show node shapes of shape:Person and shape:Student in SHACL syntax. The shape:Person has a target class :Person and a property shape for :name property with data type string and a min/max cardinality of [1,1]. The shape:Student has a target class :Student and one property shape for :regNo property with data type string and a min/max cardinality of [1,1]. And, shape:Student inherits properties and constraints of shape:Person through sh:node.

To transform those shapes into PG-Schema, a node shape with a specified sh:targetClass value is converted to a Node Type in PG-Schema. An sh:node value expressing a hierarchy is converted into the inheritance between node types using the & operator. Hence, the node shapes shape:Person and shape:Student are transformed to node types personType and studentType. The transformed PG-Schema for the node shapes in Figure 4 (a,b) is shown in Figure 5 (a,b). Property shapes expressed with sh:path instead are translated into properties or Edge Types based on their data types and cardinality constraints.

Property Shape. A node shape can have one or more properties, e.g., both node shapes shown in Figures 4a and 4b contain one property shape each, that is, :name and :regNo, respectively. The taxonomy in Figure 3 shows that a property shape defines its path using sh:path, its cardinality using sh:minCount or sh:maxCount and the kind of target node using sh:nodeKind. The cardinality constraints are converted based on both the min/max count limit they impose as well as the value of sh:NodeKind.

Cardinality Constraints. The value pair [min,max] expresses the cardinality constraints as shown in Table 1 with all possible options. We convert them into PG-Schema according to the type of node of the object to which they are connected. For example, a cardinality of [1,1] associated with a Literal target expresses a mandatory property. We transform it into a property in PG-Schema, as in the example conversion of shape:Person and shape:Student node shapes (Figure 4a, 4b) to PG-Schema (Figure 5a, 5b).

Table 1: Cardinality constraints and PG-Schema

Constraint	Description	PG-Schema Syntax
[0...]	optional property	`{OPTIONAL name: STRING ARRAY {}}`
[0...1]	optional but the maximum number of properties can be 1	`{OPTIONAL name: String }`
[0...N]	optional but the maximum number of properties can be up to N where $N > 1$	`{OPTIONAL name: STRING ARRAY {0,N}}`
[1...1]	mandatory property	`{name: String }`
[1...N]	at least one mandatory and can have up to N number of properties where $N > 1$	`{name: STRING ARRAY {1, N} }`
[M...N]	at least M mandatory and can have up to N number of properties where $M \geq 1$ and $N > 1$	`{name: STRING ARRAY {M, N} }`

Property shapes can have a single or multiple types, corresponding to literal(s) or non-literals(s). S3PG is designed to handle all kinds of nodes having single or multi-type literal or non-literal values.

Node Kind: Single Type (Literal). When transforming a single type node literal (sh:Literal), e.g., for property shapes having path :name and :regNo from Figure 4a and 4b, cardinalities strongly impact the transformation. In our example, the min/max cardinality constraints signal a mandatory property, so property shapes for :name and :regNo properties are transformed into a key value property of personType in PG-Schema (Figure 5a, 5b). Instead, a minimum cardinality value of 0 would imply an optional key-value pair property, while a max-count value >1 would require a key-value pair property where the value is an array of values. In Table 1, we provide all the possible minimum and maximum cardinality pairs with descriptions and conversion to PG-Schema when applied to a single type literal attribute such as for property shape of :name property in shape:Person node shape from Figure 4a.

Single Type (Non-Literal). A single type node can be of non-literal type defined using the sh:IRI attribute. Figure 4c illustrates the node shape of a Professor with a single non-literal type property having path :worksFor with thw expected value to be of type :Department. Such property shapes are transformed into edge types specifying the type of source and target nodes. For example, in PG-Schema, we create the edge type worksForType connecting the two corresponding converted node types, expressed using ASCII-art formatting `( )-[ ]->( )` [18]. The cardinality of the edge is translated using PG-Keys to `FOR` $p(x)$ `<qualifier>` $q(x, \bar{y})$, which allows `<qualifier>` to be expressed in the form of `COUNT <lower bound>?..<upper bound>? OF`, expressing that the number of distinct results returned by $q(x, \bar{y})$ must be within the range. For this particular property, given its cardinality [1,1], the upper and lower bounds are set to 1, as shown in the transformed PG-Schema Figure 5c.

Multi Type (Literal). Some properties may be allowed to link to literal values of multiple types. For example, the property shape for :dob property in Figure 4d can link to three types of literal values: xsd:string, xsd:date, and xsd:gYear. S3PG transforms those into node types for each possible data type, along with their IRIs, and creates an edge type with the base and target node types (as shown in Figure 5d). Therefore, the values for these literal types are encoded as node types in the target graph instead of key-value properties. This is required because arrays of property values can contain only homogeneous types. The cardinality is then translated in the same way, using PG-Keys as previously mentioned.

Multi Type (Non-Literal). Here, we illustrate transformation of property shapes that include multiple kinds of non-literal values. In Figure 4e, the property shape for the :advisedBy property can have three possible types of non-literal (sh:IRI) values, i.e., :Person, :Professor, and :Faculty. This means that a student is advised by a node that can be of type professor, faculty member, or person. This case is translated to edge types that can accept alternative

```
shape:Person
 rdf:type       sh:NodeShape;
 sh:property
     [ sh:path :name;
       sh:nodeKind sh:Literal ;
       sh:datatype xsd:string;
       sh:minCount 1;
       sh:maxCount 1   ] .
 sh:targetClass :Person.
```

(a) Person node shape with :name property shape

```
shape:Student
 rdf:type        sh:NodeShape;
 sh:property
     [ sh:path :regNo;
       sh:nodeKind sh:Literal ;
       sh:datatype xsd:string;
       sh:minCount 1;
       sh:maxCount 1   ] .
 sh:targetClass  :Student;
 sh:node         shape:Person.
```

(b) Student Node Shape with :regNo property shape

```
shape:Professor
 rdf:type       sh:NodeShape;
 sh:property
     [ sh:path :worksFor;
       sh:NodeKind sh:IRI ;
       sh:class    :Department ;
       sh:minCount 1;
       sh:maxCount 1   ] .
 sh:targetClass :Professor.
```

(c) Professor node shape with :worksFor property shape

```
shape:Person rdf:type       sh:NodeShape;
sh:property
[ sh:path     :dob ;
  sh:or
  (
      [ sh:NodeKind   sh:Literal ;
        sh:datatype   xsd:string ; ]
      [ sh:NodeKind   sh:Literal ;
        sh:datatype   xsd:date ;   ]
      [ sh:NodeKind   sh:Literal ;
        sh:datatype   xsd:gYear ;  ]
  ) ;
sh:minCount 1 ] .
sh:targetClass :Person.
```

(d) Person node shape with :dob property shape

```
shape:Student rdf:type     sh:NodeShape;
sh:property
[ sh:path     :advisedBy;
  sh:or
  (
    [ sh:NodeKind sh:IRI ;
      sh:class    :Person ; ]
    [ sh:NodeKind sh:IRI ;
      sh:class    :Professor ; ]
    [ sh:NodeKind sh:IRI ;
      sh:class    :Faculty ; ]
    ) ;
sh:minCount 1  ] .
sh:targetClass :Student .
```

(e) Student node shape with :advisedBy property shape

```
shape:GraduateStudent rdf:type sh:NodeShape;
sh:property
[ sh:path     :takesCourse ;
  sh:or
  (
   [ sh:NodeKind   sh:IRI ;
     sh:class      :Course ; ]
   [ sh:NodeKind   sh:Literal ;
     sh:datatype   xsd:string ; ]
   [ sh:NodeKind   sh:IRI ;
     sh:class      :GradCourse ; ]
   ) ;
sh:minCount 1 ] .
sh:targetClass :GraduateStudent .
```

(f) Graduate Student node shape with :takesCourse property shape

Figure 4: SHACL Shapes Examples

```
(personType:  Person {name STRING})
```

(a) Person Node type with name property

```
(studentType: Student {regNo STRING}),
(studentType: studentType & personType).
```

(b) Student Node Type with regNo property and inheriting name property from Person Node Type

```
(professorType:   Professor {} )
(departmentType:  Department {} )

(:professorType) - [worksForType: worksFor] ->
(:departmentType)

FOR (p: Professor) COUNT 1..1 OF u WITHIN (s)-
[:worksFor] -> (u: Department)
```

(c) Professor and Department Node Types with worksFor Edge Type

```
(personType: Person { name: String } )
(stringType: STRING { iri: "http:…#string"  })
(dateType:   DATE   { iri: "http:…#date"    })
(gYearType:  YEAR   { iri: "http:…#gYear"   })

CREATE EDGE TYPE (:PersonType)
-  [dobType: dob { iri: "http://x.y/dob" }]
-> (:gYearType | :stringType | :dateType)

FOR (p: Person) COUNT 1.. OF T
WITHIN (p)-[:dob]-> (T:{YEAR |STRING | DATE})
```

(d) Person, String, Date, Year Node Types with dob Edge Type

```
(personType:          Person { name: String } )
(facultyType:         Faculty )
(professorType:       Professor { name: String } )
(graduateStudentType: GraduateStudent )

CREATE EDGE TYPE (:GraduateStudentType)
-  [advisedByType: advisedBy
   { iri: "http://x.y/advisedBy" }]
-> (:personType |:professorType |:facultyType)

FOR    (g: GraduateStudent) COUNT 1.. OF T
WITHIN (g)-[:advisedBy]
  ->   (T: {Person | Professor | Faculty})
```

(e) Person, Faculty, Professor, GraduateStudent Node Types with advisedBy Edge Type

```
(personType:          Person { name: String } )
(courseType:          Course { name: String } )
(graduateCourseType:  GraduateCourse { } )
(graduateStudentType: GraduateStudent )
(stringType:          STRING {iri: "http:…#string" })

CREATE EDGE TYPE (:GraduateStudentType)
- [takesCourseType: takesCourse
  { iri: "http://x.y/takesCourse" }]
-> (:stringType | :courseType | :graduateCourseType)

FOR    (u: UnderGraduateStudent) COUNT 1.. OF T
WITHIN (u)-[:takesCourse]
  ->   (T: {String | Course | GraduateCourse})
```

(f) Person, Course, GraduateCourse, GraduateStudent, String Node Types with takesCourse Edge Type

```
(personType:  Person {name STRING}),
(studentType: Student {regNo STRING}),
(studentType: studentType & personType),
(stringType:  STRING  {iri: "http:…#string"})
(intType:     INTEGER {iri: "http:…#integer"})

CREATE EDGE TYPE (:PersonType)
- [nameType: name
  { iri: "http://x.y/name" }] -> (:stringType)

CREATE EDGE TYPE (:PersonType) - [regNoType:
regNo { iri: "http://x.y/name" }] ->
(:stringType  |  :intType)

FOR (p: Person) COUNT 1..1 OF T
WITHIN (p)-[:name]->(s: STRING)
FOR (p: Person) COUNT 1..1 OF T
WITHIN (p)-[:regNo]->(T: {STRING | INT})
```

(g) Monotone PG-Schema for Person and Student Node Types with name and regNo Edge Types

Figure 5: PG-Schema Examples

node types as targets, assuming that those target node types have already been instantiated (as shown in Figure 5e).

Multi Type (Literals & Non-Literal). Some property shapes can further allow to link at the same time to both literal and non-literal (IRI) targets. For example, the property shape for :takesCourse property in Figure 4f can take values of three types: :Course, :GradCourse, or simply xsd:string. The first two are non-literal, while the last one is of literal type. This means that a student can possibly take a course that is represented by just a title and not a proper instance of a particular class like :UnderGradCourse or :GradCourse. The transformation of such property shapes is done in a consistent manner by creating node types also for non-literal types as shown in Figure 5f in the PG-Schema defining stringType. This is required to ensure that any cardinality constraint is also properly enforced. Hence, in the transformed PG-Schema (Figure 5f) for this example has an edge type for the :takesCourse property defined using the base type graduateStudentType and target types courseType, gradCourseType, and stringType.

4.1.1 *Schema Monotonicity.*

The S3PG schema transformation described above follows a parsimonious model, as it takes advantage of cardinality constraints and encodes single value properties as key values within nodes anytime that is possible. Such parsimonious transformation is suitable for graphs with schemas that do not experience structural changes or that always have single types or homogeneous properties. *As described below, when the schema does not change, our graph data transformation (which introduce below) is monotonic by construction.* However, when dealing with graphs having schemas that often evolve and go through structural modifications, such as the addition or removal of properties and change of data types of properties, this solution will not be able to adapt and thus preserve monotonicity when, for instance, single-value properties become multi-value properties during evolution. That is, we may requite to re-convert properties of nodes already converted. To support graphs that have an evolving schema, we also propose an alternative non-parsimonious model to ensure that the transformation is *schema monotone*.

To address changes that would break monotonicity, S3PG follows a non-parsimonious approach and models all properties as multi-type literal and non-literal properties as described above. We call this type of transformation *non-parsimonious* because it forces to instantiate both edges and nodes also for key-value pairs that could instead have been simply encoded as properties within a node record. We use the shape schema in Figure 4b as an example to demonstrate this. Assume that the property shape for :regNo of node shape shape:Student allows values of type xsd:integer in addition to strings. In this case, S3PG non-parsimonious transformation models all properties of type Person and Student node shape as *edge types*, allowing for the addition of more data types for the target node type in udpates happening after a first transformation process has concluded. The monotonic PG-Schema for the shape schema of Figure 4b is presented in Figure 5g.

4.2 Data Transformation

Similar as for the schema transformation, S3PG offers two alternatives of the data transformation as well: *parsimonious* and *non-parsimonious*. The main difference between both types of transformation approaches lies in the way single value properties are modeled in the PG data model. In case of the parsimonious approach, depending on the cardinalities of the properties, S3PG tries to represents the data as key value attributes within nodes whenever possible. In case of the non-parsimonious approach, S3PG opts for modeling all properties data as nodes to accommodate structural changes in the future, and in doing so preserves monotonicity of the transformation.

Transforming an RDF graph G into a property graph PG using PG-Schema $\mathcal{S}_{PG}$ (created by transforming the shape schema $\mathcal{S}_G$ of G in Section 4.1) requires processing its triples and analyzing the types of nodes involved both as subjects and objects in those triples. At a high level, at first, we need to know all the entities and their types and then their property information to create PG conforming to $\mathcal{S}_{PG}$. We propose a two-phase algorithm as a solution to Problem 2 in Algorithm 1, which in the first step extracts entities and transforms them into nodes in PG, and then parses the property related information to create edges and attributes in PG.

The algorithm takes G in the form of file **F** and the PG-Schema $\mathcal{S}_{PG}$ as inputs and reads **F** triple by triple to process the stream of triples $\langle s, p, o\rangle$ (Line 4 ff.). In the first phase, it extracts entities and their types into the Ψ_{etd} map (Lines 6-7) and then iterates over entities in Ψ_{etd} to create PG nodes (Line 8 ff.). This is done by initializing nodes (Line 9) by assigning labels as types of entity $e.\mathrm{T}$ such that each node and its assigned labels comply with N_s in $\mathcal{S}_{PG}$ (Line 11), encoding the IRI of each entity as a key value within the node (Line 13), and adding the node to the set of nodes N in PG (Line 14). Considering the RDF graph in Figure 2a, the algorithm parses entities :bob, :alice, and :DB with their types {:Person, :Student, :GS}, {:Person, :Faculty, :Professor}, and {:Course, :GC}, and creates PG nodes with labels and types, as shown in Figure 2c.

Now that the nodes $N \in PG$ are created using the entity data from G, in the second phase the algorithm extracts the property data of each entity to create edges $E \in PG$. Specifically, it iterates over triples $\langle s, p, o\rangle \in G$ (Line 15 ff.), such that for every non-type triple t, initiall, it evaluates if $t.o$ exists as node type in the set of nodes $N \in PG$, which means that $t.o$ is a non-literal value (IRI) having type T in $N_s \in \mathcal{S}_{PG}$ (Line 16). In affirmative cases, it uses the function $\rho \in PG$ to create an edge between nodes corresponding to $t.s$ and $t.o$ in N. Hence, it uses $\mathcal{S}_{PG}$ to find out how $t.p$ edge should be modeled in PG, i.e., if node labels $t.s$ and $t.o$ are in the set $\mathcal{L}$ specified by $t.p$ edge in $\mathcal{S}_{PG}$. If $\mathcal{S}_{PG}$ assures compliance of both source $t.s$ and target $t.o$ nodes for $t.p$, an edge is initialized for $t.p$ on line 19 using function ρ (Line 20). Continuing with the example, the algorithm now extracts non-type triples t_1: ⟨:bob, :regNo, 'Bs12'⟩ and t_2: ⟨:bob, :advisedBy, :alice⟩ and parses object values classifying them as literals or IRIs. For IRIs such as :alice in t_2, it creates an edge between PG nodes associated with the subject and object values using the predicate :advisedBy. For literals, such as 'Bs12' in t_1, the algorithm checks the data type and cardinality before proceeding to the next steps to create key values or edges in PG.

Next, it evaluates the data type and cardinality of $t.p$ edge having label $L(t.s)$ for source node (Line 21). If it is a literal and has a minimum cardinality 0 or 1 and a maximum cardinality 1, then $t.p$ is initialized as a key value edge and the function π is used to assign its record value to the node corresponding to $t.s$ based on the data type of the literal value such that the type of $t.s$ (as a label) exists in $\mathcal{L}$ (Lines 22-23). Lastly, the only remaining choice for $t.p$ can either be a multi-type homogeneous property (consisting of literals or non-literals) or a heterogeneous property (containing both literals and non-literals) values – as depicted in Figure 3 with node kinds. To deal with such cases, we create an edge of the appropriate type that corresponds to a PG node with a literal label as specified in PG-Schema. Specifically, the data type and value are extracted from $t.o$ and a PG node is initialized (Lines 25-27), the label is assigned using the extracted data type in conformance with $\mathcal{S}_{PG}$ (Line 28), the value is encoded as a key value attribute within the created node (Line 30), and finally the source node is linked to the target node using the ρ function of PG (Line 31).

4.2.1 ***Graph Monotonicity.*** The S3PG data transformation algorithm section follows parsimonious model. In order to preserve the monotonic transformation property, S3PG follows non-parsimonious model to transform RDF data into PG model. In the second phase, Algorithm 1 (Lines 21–23), instead of storing literal properties having cardinality (0,1) or (1,1) as key value properties within nodes in PG, the non-parsimonious approach models them similar to the way other properties are modeled (Lines 25–31). This way, non-parsimonious model provides full monotonicity w.r.t. any structural

Algorithm 1 S3PG: DATA TRANSFORMATION

```
    Input: Graph $G$ from File F, $\mathcal{S}_{PG} = (N_S, E_S, \nu_S, \eta_S, \gamma_S)$
    Output: Property Graph $PG = (N, E, \rho, \lambda, \pi)$
 1: $\Psi_{\text{ETD}}$ = Map⟨T: SET_Types⟩
 2: $N$ = [ ] // PG Nodes
 3: $E$ = [ ] // PG Edges
 4: for $t \in G$ ∧ t.p = Type Predicate do          ▷ $t$ < $s$:subject, $p$:predicate, $o$:object >
 5:     $e$ : t.s // entity , $e_t$ = t.o // entityType
 6:     if $e \notin \Psi_{\text{ETD}}$ then $\Psi_{\text{ETD}}$.insert($e$, ... ))
 7:     $\Psi_{\text{ETD}}$.insert($e$, $\Psi_{\text{ETD}}$.get($e$).T.add($e_t$))          ▷ T : entity types
                                                          ▷ ① Entities to PG Nodes
 8: for $e \in \Psi_{\text{ETD}}$ do
 9:     node = initNode( )
10:     // Function to add labels to nodes : $\lambda : (N \cup E) \rightarrow 2^{\mathcal{L}}$ (Definition 2.4)
11:     $\lambda$(node, $\mathcal{L}$ : $e$.T) s.t. $e$.T $\models (N_s \in \mathcal{S}_{PG})$          ▷ $e$.T $\in \mathcal{L}$
12:     // Function to map nodes & edges to records: $\pi : (N \cup E) \rightarrow \mathcal{R}$ (Definition 2.4)
13:     $\pi$(node, initPgKey($iri$)) $\rightarrow$ $e$.IRI
14:     N.add(node)
                                          ▷ ② Properties to PG KeyValues and Edges
15: for $t \in G$ ∧ t.p != Type Predicate do
16:     if $t.o \in \Psi_{\text{ETD}}$ then                   ▷ $t.o$ exists as a node type
17:         // Function to map edge to nodes: $\rho : E \rightarrow (N \times N)$
18:         if $\exists\, L$(N(t.s) ∧ N(t.o)) $\in \mathcal{L}$ then
19:             $edge$ =initPgEdge($t.p$ )                     ▷ $edge \in E_s$
20:             $\rho(E : edge$, ($N$(t.s), $N$(t.o) )   ▷ { $\mathcal{T}$ ($N$(t.s)) & $\mathcal{T}$ ($N$(t.o)) } $\in N_s$
21:     else if $isLiteral(E_s(t.p)$, $L$(N(t.s)) && $card(E_s(t.p))$ ∈[0|1,1]) then
22:         $edge$ = initPgKey($t.p$)
23:         if $\exists\, L$(N(t.s)) $\in \mathcal{L}$ then $\pi$( $N$(t.s), $edge$) $\rightarrow t.o$
24:     else
25:         $O_{type}$ = extractDataType($t.o$)
26:         $O_{value}$ = extractValue($t.o$)
27:         node = initNode( )
28:         $\lambda$(node, $\mathcal{L}$ : $O_{type}$) s.t. $O_{type} \models \{(N_s \in \mathcal{S}_{PG})\}$     ▷ $O_{type} \in \mathcal{L}$
29:         $edge$ =initPgEdge($t.p$)                         ▷ $edge \in E_s$
30:         $\pi$(node, initPgKey($value$)) $\rightarrow O_{value}$
31:         if $\exists\, L$(N(t.s)) $\in \mathcal{L}$) then $\rho(E : edge$, ($N$(t.s), node )
```

changes in the schema and adapts to such changes during data transformation.

4.2.2 ***Complexity Analysis.*** The time complexity of S3PG's data transformation (Algorithm 1) depends on the size of the input RDF graph, denoted as $|F|$ representing the number of triples in G. The number of entities extracted from the graph $|N|$, and the size of the set of allowed labels $\mathcal{L}$ in the PG. The initialization of data structures takes constant time. Iterating over the triples in the input graph for entity extraction and property transformation takes $O(|F|)$ time, while transforming entities into property graph nodes takes $O(|N|)$ time. The time complexity for the property transformation, involving checks for label existence and mapping edges to nodes, is $O(|F| \cdot L)$. Consequently, the overall time complexity of the algorithm is $O(|F| + |N| + |F| \cdot L)$.

4.3 Transformation Properties

In the following, we show that S3PG preserves the transformation properties defined in Section 3.

PROPOSITION 4.1. *SP3G data and schema transformations are information preserving.*

Showing this proposition to be true involves providing computable mappings [37] $\mathcal{N}(\mathcal{S}_G)$ and $\mathcal{M}(G)$ that satisfy the condition in Definition 3.1, that is, a computable mapping $\mathcal{N}(\mathcal{S}_G)$ that can reconstruct the initial shape schema from the transformed PG-SCHEMA and $\mathcal{M}(G)$ that can reconstruct the initial RDF graph from the transformed property graph. This is always possible since our schema transformation rules (Section 4.1) are invertible and the data transformation process is complete, i.e., no information is discarded during transformation, and the process is non-ambiguous, i.e., it does preserve node identifiers and it is not possible for two different data items to create the same node, edge, or property.

PROPOSITION 4.2. *SP3G transformations are semantics preserving.*

The validity of this proposition is based on the semantic preservation properties of the schema transformation function F_{st} and then of the data transformation function F_{dt}. At the base of this proposition is the fact that our transformation rules identify a way to encode each and all the relevant RDF shape constraints into equivalent PG schema constraints.

Given $\mathcal{S}_G$ having a set of node shapes $\langle s, \tau_s, \Phi_s \rangle \in \mathcal{S}_G$ and $\mathcal{S}_{PG}$ for the output PG. The schema transformation function F_{st} transforms $\mathcal{S}_G$ into $\mathcal{S}_{PG}$: $F_{st}(\mathcal{S}_G) = \mathcal{S}_{PG}$. Consider an arbitrary property shape constraint C of a node shape in $\mathcal{S}_G$. By definition, $F_{st}(C)$ maps C to a corresponding constraint in $\mathcal{S}_{PG}$. We aim to prove that $F_{st}(C)$ preserves the semantics of C. Let I be any RDF data instance such that $I \models C$ for C in $\mathcal{S}_G$. We need to show that $F_{st}(I) \models F_{st}(C)$ in the PG-SCHEMA ($\mathcal{S}_{PG}$). Since $I \models C$, it implies that C is satisfied by I according to the semantics of $\mathcal{S}_G$ (see Definition 2.3). By the definition of F_{st}, $F_{st}(I)$ should satisfy $F_{st}(C)$ in $\mathcal{S}_{PG}$ to preserve semantics. This shows that, for any instance $I \models C$, $F_{st}(I) \models F_{st}(C)$ in $\mathcal{S}_{PG}$, preserving the semantics of the individual constraints. Therefore, the entire $\mathcal{S}_{PG}$ preserves the semantics of $\mathcal{S}_G$.

We then establish in a constructive way that both transformations F_{st} and F_{dt} are semantics preserving. Given a valid RDF graph instance $I \in G$ where $I \models \mathcal{S}_G$, we have the data transformation function $F_{dt}(I)=P$, and $P \in PG$. Then, having $F_{dt}(I)=P$ we obtain P that conforms to the constraints specified in $\mathcal{S}_{PG}$, i.e., $P \models \mathcal{S}_{PG}$. Let C_{pg} represent any constraint in $\mathcal{S}_{PG}$, and let C_{shape} represent the corresponding constraint in $\mathcal{S}_G$ based on the schema transformation (F_{st}) mapping. Now, consider an instance $I' \in P$ that satisfies C_{pg}. We need to show that I' also satisfies C_{shape}. Since $I' \models C_{pg}$, it implies that C_{pg} is satisfied by I' according to the semantics defined in $\mathcal{S}_{PG}$ (see Definition 2.3). By the definition of F_{st}, C_{pg} corresponds to C_{shape} in $\mathcal{S}_G$. Therefore, I' satisfies C_{shape} as well in the original RDF graph G. Since this holds true for any constraint C_{pg} in $\mathcal{S}_{PG}$, it follows that P conforms to the constraints specified in $\mathcal{S}_{PG}$: $P \models \mathcal{S}_{PG}$. Hence, the data transformation method $F_{dt}(I)$ is by construction designed to preserve the semantics of the RDF graph instances, ensuring that the resulting property graph P conforms to the constraints in $\mathcal{S}_{PG}$.

PROPOSITION 4.3. *SP3G transformations are monotonic.*

Monotonicity refers to the property that incremental changes to the RDF graph and its schema result in consistent and predictable updates to the property graph and its schema, preserving existing data and structure without requiring the re-computation of the entire transformation [37]. Our implementation is monotonic because: if the schema $\mathcal{S}_G$ does not change (thus also the cardinality constraints remain the same), we incrementally convert the data in G_Δ, by only adding new edges, new attributes, or new attribute values. This ensures that $F_{dt}(G')=F_{dt}(G) \Cup F_{dt}(G_\Delta)$, where $G'=G \Cup G_\Delta$ represents the RDF graph after applying changes G_Δ. When new types and constraints are added to $\mathcal{S}_G$, e.g., new property shapes, the non-parsimonious encoding allows us to add new constraints to $\mathcal{S}_G$ and update only the cardinality constraints involved in the update, without altering all the others. This ensures that, in addition to the above, we also have $F_{st}(\mathcal{S}'_G)$=$F_{st}(\mathcal{S}_G) \Cup F_{st}(\mathcal{S}_{G_\Delta})$, where $\mathcal{S}'_G$=$\mathcal{S}_G \Cup \mathcal{S}_{G_\Delta}$ represents the shape schema after applying changes $\mathcal{S}_{G_\Delta}$. This is fundamental for handling heterogeneous literal and non-literal attributes as they do not require to delete existing attributes but only add the new ones with edges and nodes.

Once the transformation of G into PG is completed, we can also transform queries over G. Let F_{qt} be a query translator function to translate a SPARQL query Q over G into Q' over PG. In S3PG, F_{qt} can make use of $\mathcal{S}_{PG}$ to translate Q into Q' as $PG \models \mathcal{S}_{PG}$. We use various queries on DBpedia [6] in Section 5 to empirically demonstrate that the results R obtained by executing an example query Q over G and R' obtained by executing a translated query Q' over PG are identical, meaning that $R \subseteq R'$ and $R' \subseteq R$.

The correctness of S3PG is established by proving its information preservation and semantics preservation properties. *Information preservation* is ensured through computable mappings $\mathcal{N}(\mathcal{S}_G)$ and $\mathcal{M}(G)$, along with F_{st}, which ensures the reconstruction of the initial shape schema from the transformed PG-Schema and F_{dt}, which ensures the reconstruction of the initial RDF graph from the transformed PG. *Semantics preservation* is demonstrated by showing that both schema and data transformation methods maintain the semantics of the original RDF graph and shape schema. Thanks to the fact that any RDF graph can be mapped to an equivalent property graph, and thanks to the mapping we established between the SHACL schema and the PG schema, our conversion preserves such properties. Further, it offers guidance on how to convert queries between data models to enable testing query equivalence.

5 EXPERIMENTAL EVALUATION

We assess the effectiveness of S3PG and demonstrate its effectiveness and efficiency compared to existing methods.

Datasets. We chose two versions of DBpedia [6] (one from December 2022 and an earlier version from 2020) and the Bio2RDF Clinical Trials (CT) [8] dataset for experimentation. The DBpedia datasets containing up to 332 million triples allow us to test the scalability of S3PG and Bio2RDF CT dataset containing 132 million triples allows us to thoroughly explore a domain-specific RDF graph. The statistics and characteristics of these notably very large and non-trivial datasets are presented in Table 2.

Experimental Setup. S3PG is implemented in Java-17 (source code and experimental settings are available at https://github.com/dkw-aau/s3pg). We conducted all experiments on a single machine with 16 cores and 256 GB RAM, running Ubuntu 18.04. Among existing SHACL extraction techniques [9, 13, 17, 25, 30–33] from RDF graphs, we used [33] to extract SHACL shapes for each dataset to transform them into PG-Schema. We compare S3PG with the two state-of-the-art graph transformation approaches: NeoSemantics [28] and rdf2pg [5], which output PGs as Neo4j [26] instance.

Metrics. We evaluate the performance of S3PG by measuring the time it takes for schema and data conversion, the maximum memory usage, and the time needed to load the transformed graph into a PG DBMS. We further demonstrate the necessity of the S3PG transformation technique by evaluating the quality of the PGs created using S3PG and other methods on DBpedia2022. This evaluation involves executing a set of queries on both the original and transformed graphs and then measuring the accuracy based on the completeness of the query results.

We also analyze the effect of the transformation from RDF to PG on query runtime by executing these queries on the source RDF graphs and transformed PGs. Additionally, we demonstrate that S3PG is monotonic with respect to schema and graph changes by using the two DBpedia datasets. Finally, we show that S3PG preserves the semantics, i.e., all the integrity constraints, by executing a set of queries to ensure that the same integrity constraints hold over both the source and transformed graphs.

5.1 Transformation Analysis

We perform an analysis of the transformations of the RDF graph datasets using three distinct techniques: S3PG, NeoSemantics [28], and rdf2pg [5]. Table 4 presents the transformation and loading times for these techniques applied to the aforementioned datasets.

NeoSemantics. NeoSemantics is an extension of the Neo4j database, implemented in Java. It enables users to configure various transformation parameters, such as how to handle multivalue properties, vocabulary URIs, and RDF types during transformation [27]. NeoSemantics leverages Neo4j's internal libraries to transform and map data by loading it into Neo4j. Consequently, when using NeoSemantics, it is not possible to differentiate between the transformation and loading times. As depicted in Table 4, NeoSemantics successfully transformed and loaded DBpedia2020 in 38 minutes, DBpedia2022 in 5.5 hours, and Bio2RDF CT in 1.3 hours. All these transformations adhere to a 32 GB memory limit.

rdf2pg. The rdf2pg transformation approach offers three distinct mapping types for converting an RDF graph to a PG: simple instance mapping, general database mapping (schema-independent), and direct database mapping (schema-dependent). Here we considered the schema-dependent direct mapping variant. It is implemented in Java and its source code is accessible on GitHub [34]. It outputs PG graphs in YARS-PG [39] serialization format, and it also provides a Neo4JWriter module for transforming it into a set of Cypher queries for loading into Neo4j. However, loading a property graph into Neo4j using Cypher queries is only suitable for small graphs. To mitigate this, we enhanced Neo4JWriter to produce the graph in CSV format, which significantly improved its loading efficiency. As outlined in Table 4, rdf2pg accomplished the transformation and loading of DBpedia2020 in 23 minutes, DBpedia2022 in 2.6 hours, and Bio2RDF CT in 1.1 hours. Notably,

Table 2: Size and characteristics of the datasets

	DBpedia 2020	DBpedia2022	Bio2RDF CT
# of triples	52 M	332 M	132 M
# of objects	19 M	101 M	54 M
# of subjects	15 M	34 M	10 M
# of literals	15 M	49 M	48 M
# of instances	5 M	22 M	10 M
# of classes	427	775	65
# of properties	1,323	51,146	168
Size in GBs	6.6	45	25

Table 3: SHACL Shapes Statistics of DBpedia (2020 & 2022) and Bio2RDF datasets

	# of NS	# of PS	# of Single Type PS	# of Multi Type PS	Single Type PS		Multi Type Homo PS		Multi Type Hetero PS
					Literals	Non-Literals	Literals	Non-Literals	Literals & Non-Literals
DBpedia 2022	746	622,237	206,735	415,502	383,355	14,830	75,129	31,563	100,043
DBpedia 2020	426	12,354	3,452	8,902	5,337	2,069	0	3,452	0
Bio2RDF CT	65	891	292	599	387	64	93	196	3

Table 4: Analysis of Transformation (T) and Loading (L) times in minutes (m) and hours (h)

	DBpedia 2020			DBpedia2022			Bio2RDF CT		
	T	L	*Sum*	T	L	*Sum*	T	L	*Sum*
S3PG	3 m	9 m	*12 m*	34 m	1.2 h	*1.7 h*	16 m	26 m	*42 m*
rdf2pg	15 m	8 m	*23 m*	1.6 h	58 m	*2.6 h*	45 m	21 m	*1.1 h*
NeoSem	-	-	*38 m*	-	-	*5.5 h*	-	-	*1.3 h*

rdf2pg required more RAM compared to NeoSemantics due to its in-memory transformations. Nevertheless, all transformations were completed within a maximum memory limit of 64 GB.

S3PG. The S3PG algorithm proved to be more effective than other methods in the execution of schema and data transformations, mainly due to its two-pass data transformation algorithm. When considering the overall transformation and loading time, S3PG achieved better results by transforming and loading DBpedia2020 in just 12 minutes, DBpedia2022 in 1.7 hours, and Bio2RDF CT in 42 minutes. Similar to NeoSemantics, S3PG successfully conducted all transformations while staying within a 32 GB memory limit.

Statistical Analysis of the Transformed Graphs. The PGs produced by S3PG contain 50% more nodes and edges than those transformed with NeoSemantics and rdf2pg (Table 5). These numbers are expected given the shapes statistics in Table 3, which shows that DBpedia2022 contains more than 75k multi-type homogenous literals and 100k multi-type heterogeneous property shapes. S3PG models values of such properties as nodes instead of treating them as strings and storing them in an array as key values, which leads to the creation of a larger number of nodes and edges.

5.2 Quality Analysis

We assess the quality of the PGs created by S3PG and other existing transformation methods by evaluating the precision of responses to a series of queries on DBpedia2022 and Bio2RDF. We formulated

Table 5: Transformed Graphs (PG models) Stats: M denotes millions; NeoSem abbreviates NoeSemantics.

		# of Nodes	# of Edges	# of Rel Types
DBpedia 2022	S3PG	120 M	157 M	18,216
	NeoSem	54 M	129 M	12,922
	rdf2pg	54 M	129 M	12,922
DBpedia 2020	S3PG	26 M	27 M	655
	NeoSem	16 M	18 M	632
	rdf2pg	16 M	18 M	632
Bio2RDF CT	S3PG	40 M	59 M	103
	NeoSem	10 M	29 M	64
	rdf2pg	10 M	29 M	64

SPARQL queries for the RDF graphs and then, using the mapping F_{ts} generated by the algorithms as reference, converted them *manually* into Cypher queries for each transformation method. At present, developing an automated query translator falls beyond the scope of this work. We divided the queries into four distinct categories based on the categorization of node shape constraints from Figure 3. These categories include *single-type* queries, *multi-type homogeneous literal* queries, *multi-type homogeneous non-literal* queries, and *multi-type heterogeneous (literal and non-literal)* queries. For DBpedia2022, we created a total of 120 queries, with 60 queries for each of the first three categories and 60 queries for the last category. For Bio2RDF we created a total of 36 queries, with 27 queries for first three categories and 9 queries for the last category.

We executed the SPARQL queries over the RDF graphs and computed the number of results for each query as ground truth. Then we compared the results returned by Cypher queries executed over the PGs created by S3PG, NeoSemantics, and rdf2pg. Tables 6 and 7 show the ground truth (# of GT) and accuracy (in percentage) for all queries on DBpedia2022 and Bio2RDF. A higher percentage indicates a more accurate or complete transformation of the RDF graph into a PG for data corresponding to that specific type of query. Therefore, when we state that S3PG achieved 100% accuracy for a particular query, it means that it produced the exact expected result count, preserving all the information. In contrast, when the percentage is lower, e.g., for NeoSemantics (NeoSem) and rdf2pg, it indicates that these methods produced results that differ from the expected results, suggesting a lossy transformation. S3PG consistently achieves 100% accuracy, indicating that it preserves all information as discussed above.

Accuracy Analysis on DBpedia2022. For single-type queries (Q1–Q5), NeoSem shows high accuracy while rdf2pg's accuracy ranges from 99.45% to 100%. For multi-type homogeneous (MT-Homo) literals (L) queries (Q6–Q10), NeoSem shows high accuracy, and rdf2pg's accuracy ranges from 84.62% to 100%. This indicates that S3PG and NeoSem are both more reliable for MT-Homo (L) queries than rdf2pg. For MT-Homo non-literal (NL) queries (Q11–Q15), all methods achieve 100% accuracy in our queries. For MT-Hetero (L+NL) queries (Q16–Q25), NeoSem's accuracy ranges from 90.48% to 99.99%, while rdf2pg's accuracy varies widely, from 30.22% to 99.99%. For example, we can analyze Q22 from Table 6. Below, we present its SPARQL and Cypher variants, transformed according to S3PG and NeoSemantics.

```
Q22: SPARQL:
SELECT ?e ?p WHERE { ?e a schema:ShoppingCenter ; dbp:address ?p .}

Q22: S3PG:
MATCH (n:sch_ShoppingCenter)-[:dbp_address]->(tn)
RETURN n.iri AS node_iri, COALESCE(tn.ov, tn.iri) AS tn_iri_or_value;
```

```
Q22: NeoSemantics:
MATCH (node:sch_ShoppingCenter)-[:sch_address]->(tn)
RETURN node.uri AS node_uri, tn.uri AS tn_iri_or_value
UNION ALL
MATCH (node:sch_ShoppingCenter)
UNWIND node.sch_address AS tn_iri_or_value
RETURN node.uri AS node_uri, tn_iri_or_value;
```

Hence, compared to the corresponding SPARQL query, the S3PG Cypher version of Q22 produces the complete result, while NeoSemantics (and similarly rdf2PG) produces an incomplete result. This is because the values of `dbp:address` in the graph are sometimes of type `STRING`, sometimes of type `INTEGER`, and sometimes IRIs – only S3PG is able to correctly map this case.

Accuracy Analysis on Bio2RDF. For single-type queries (Q1–Q3), NeoSem shows high accuracy while rdf2pg's accuracy ranges from 99.71% to 100%. For multi-type homogeneous (MT-Homo) literals (L) queries (Q4–Q6), NeoSem shows high accuracy, and rdf2pg's accuracy ranges from 99.73% to 99.77%. For MT-Homo non-literal (NL) queries (Q7–Q9), all methods achieve 100% accuracy. For MT-Hetero (L+NL) queries (Q10–Q12), NeoSem's accuracy is 99.99%, while rdf2pg's accuracy varies from 97.89% to 99.99%.

These experiments not only prove that NeoSem and rdf2pg cannot offer a reliable transformation but also that the loss of data can have very dramatic repercussions on query completeness.

5.3 Effect on Query Runtime

We study how query runtime varies between the original RDF graphs and the transformed PGs. The goal is to show that our proposed transformation method does not create a schema that is now too complex to query. The runtimes measured are highly dependent on the Graph DBMS adopted, therefore this exploratory experiment is not designed to conclude anything on the general PG vs. RDF performance. We executed queries on DBpedia2022 RDF as well as the corresponding PGs transformed by S3PG, NeoSemantics, and rdf2pg. We used GraphDB (9.9.0) to execute SPARQL queries and 1000 warm-up queries before executing the benchmark queries. For Neo4j, we used the CALL apoc.warmup.run() function to warm up. We ran each query 10 times and measured the average runtime. Figure 6 presents the results based on our four query types. Overall, we see that query runtimes remain comparable between the different data models and transformation techniques. The most significant difference is for S3PG in multi-type heterogeneous queries (Q16-Q30), where its direct modeling of multi-value and multi-type properties' data in nodes allows it to improve some query execution times compared to NeoSemantics and rdf2pg. Instead, for some queries in Figure 6(d), S3PG takes longer than NeoSemantics and rdf2pg. The reason is that, since S3PG is a fully data preserving transformation technique supporting multi-type heterogeneous literals and non-literals queries, it retrieves more data (the correct complete result) in contrast to the other two techniques.

Notably, SPARQL queries on GraphDB prove effective for both single and multi-type queries. Nevertheless, in multi-type heterogeneous queries (Q16-Q30), SPARQL appears to be less efficient than S3PG's Cypher queries. We stress that this experiment does not aim to provide a comparison between different system architecture performances, which requires a different type of analysis [24], while confirming that query performance ought to be considered when deciding which model to adopt.

Table 6: Accuracy analysis (in percentages) for DBpedia2022 – PG and RDF, transformed by S3PG, NeoSemantics, rdf2pg

		# of GT	S3PG	NeoSem	rdf2pg			# of GT	S3PG	NeoSem	rdf2pg
Single Type	Q1	1,200,712	100%	100%	100%	MT-Hetero (L+NL)	Q16	210,003	100%	99.97%	45.08%
	Q2	282,358	100%	100%	99.46%		Q17	98,595	100%	99.78%	78.66%
	Q3	89,880	100%	100%	99.45%		Q18	93,586	100%	99.99%	79.06%
	Q4	80129.00	100%	100%	99.49%		Q19	1,603	100%	99.69%	99.06%
	Q5	10.00	100%	100%	100%		Q20	11,969	100%	97.76%	91.15%
MT-Homo (L)	Q6	22,566	100%	100%	99.06%		Q21	924	100%	90.48%	79.55%
	Q7	5,509	100%	100%	99.07%		**Q22**	1,831	100%	92.63%	89.30%
	Q8	13	100%	100%	84.62%		Q23	5	100%	100%	100%
	Q9	3	100%	100%	100%		Q24	48,146	100%	94.09%	91.02%
	Q10	52	100%	100%	x		Q25	376	100%	98.14%	94.95%
MT- Homo (NL)	Q11	1,439,679	100%	100%	100%		Q26	13,628	100%	97.23%	87.40%
	Q12	13,111	100%	100%	100%		Q27	687	100%	99.85%	99.27%
	Q13	318,414	100%	100%	100%		Q28	141,570	100%	99.72%	98.88%
	Q14	11	100%	100%	100%		Q29	31,123	100%	99.99%	30.22%
	Q15	55	100%	100%	100%		Q30	7	100%	100%	57.14%

Table 7: Accuracy analysis (in percentages) for Bio2RDF – PG and RDF, transformed by S3PG, NeoSemantics, rdf2pg

		# of GT	S3PG	NeoSem	rdf2pg			# of GT	S3PG	NeoSem	rdf2pg
ST	Q1	163,894	100%	100%	99.71%	MTH-NL	Q7	195,545	100%	100%	100%
	Q2	18,089	100%	100%	99.81%		Q8	1,089,005	100%	100%	100%
	Q3	1,279	100%	100%	100%		Q9	798,552	100%	100%	100%
MTH-L	Q4	30,752	100%	100%	99.77%	$\text{MT}_{\text{Hetro}}^{\text{L+NL}}$	Q10	1,254,745	100%	99.99%	99.99%
	Q5	19,530	100%	100%	99.77%		Q11	1,095,319	100%	99.99%	99.79%
	Q6	42,726	100%	100%	99.73%		Q12	2,465,643	100%	99.99%	97.89%

5.4 Monotonicity Analysis

As mentioned above, S3PG offers two types of transformation approaches: parsimonious and non-parsimonious. For evolving graphs, the non-parsimonious model allows schema evolution for different attribute types. To study the monotonicity properties, we leverage two snapshots of DBpedia: the snapshot from December 2022 (Dbp22march), and another snapshot from March 2022 (Dbp22dec). The Δ between both snapshots shows the addition of 16.7M new triples and the deletion of 5.9M triples. We consider as updated triples all those triples with changes in their object values only; these amounts to 16.1$million$ updated triples. Approximately 1.84% of the triples were deleted, and 5.21% of the triples were added from March (Dbp22march) to December (Dbp22dec).

When employing the *parsimonious* transformation approach, we use S3PG to transform Dbp22march and Dbp22dec and measure the transformation time. Converting Dbp22march took 34.0 minutes with the parsimonious transformation model and 31.83 minutes with the non-parsimonious transformation model, while converting Dbp22dec from scratch with the parsimonious model took 34.25 minutes, i.e., approximately 7.59% longer than using the non-parsimonious model. Instead, converting only Δ with the non-parsimonious model took 9.97 minutes. This indicates that adopting the non-parsimonious model to incorporate Δ in Dbp22march results in a time savings of 24.28 minutes, representing a substantial 70.87% reduction in overall transformation time. Thus, even though the data model is slightly more complex the incremental transformation is more efficient.

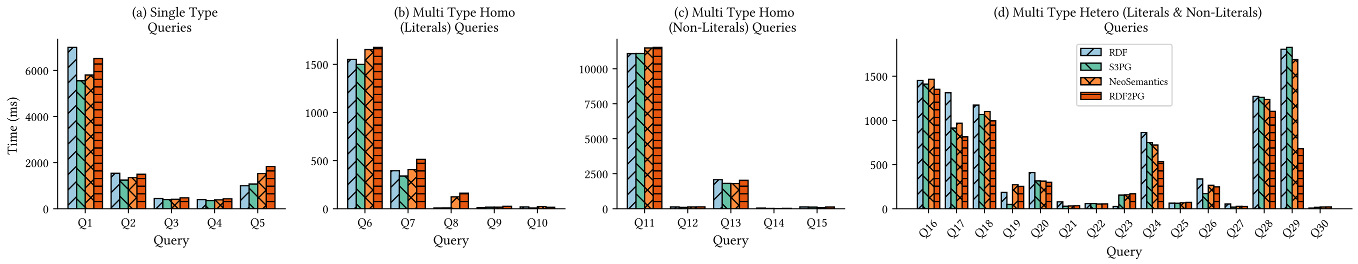


Figure 6: Query runtime analysis on DBpedia2022

6 RELATED WORK

Few approaches have been proposed to tackle the interoperability between RDF graphs and the PGs.

Rdf2pg [5] proposes a direct mapping approach for transforming RDF graphs into PGs covering both data and schema. This approach provides both schema-dependent and schema-independent direct mappings for the transformation process. It utilizes RDFS as the schema for KGs and a custom-defined schema for PGs. The experimental results of rdf2pg conclude that PGs subsume the information capacity of RDF graphs. Contrary to this approach, our proposed method uses current standards – SHACL schema for RDF graphs and PG-Schema [1] for PGs – which allow defining node and edge types, and support advanced scenarios such as expressing complex type hierarchies and integrity constraints. We show that using these schema languages for transformation is a better and more reliable approach to ensure semantics and information preservation of schema and data in graphs. G2GML [12] is a language for mapping RDF graphs to PGs. G2GML transformations are manually specified *ad hoc* by users for a given dataset. Unlike S3PG, Rdf2pg, and Neosemantics, G2GML is just a mapping language, thus it does not provide an automatic transformation algorithm. It is possible to extend S3PG to produce G2GML mappings as an additional output though.

One of the issues with transforming RDF graphs into PGs is that one almost always ends up with data that is not complete after transformation [16]. In this direction, Hugegraph [19] proposed an approach to transform RDF to PGs which addresses several challenges that arise during the conversion process, such as ensuring the uniqueness of nodes and supporting multi-label and empty-label RDF graphs in single-label graph databases. In their work, the authors focused on converting RDF graphs into Hugegraph, which is a specific graph database management system; this approach does not apply to other graph databases in general. Furthermore, the authors stated that this method is not guaranteed to be information preserving because, while importing RDF graphs into Hugegraph, the system tries to automatically identify the labels of vertices and edges and also omits blank nodes along with all of their related edges. An alternative method is implemented in rdf2neo [11], a tool that allows to populate Neo4j databases from RDF graphs, and uses real use cases from agrigenomics to demonstrate how this tool can be used to increase opportunities for knowledge sharing and interoperability. In contrast to S3PG, it is not schema-based and does not guarantee completeness. In the work by Zhang et al. [44], a method using a mapping structure is introduced as a advancement in achieving a bidirectional mapping between RDF and PGs. Unlike S3PG, this method does not rely on a schema-based approach to guarantee the completeness and monotonicity of the transformation from RDF to PGs.

Some other works [3, 45] propose a unified storage layout for RDF graphs and PGs. However, there is currently still a need for a unifying data model that can support graph formats like RDF, RDF-star, and PGs, while at the same time being powerful enough to naturally store information from complex knowledge graphs, such as Wikidata, without the need for a complex reification scheme [3]. Our approach is the first that can effectively take advantage of the semantic information encoded in the SHACL to obtain an equivalent PG-schema thus guaranteeing a complete conversion from RDF to PGs. The opposite direction can take advantage of our methodology but requires a different treatment.

7 CONCLUSION

S3PG is a method for transforming RDF graphs into Property Graphs (PGs) based on standardized SHACL shapes. S3PG represents a schema-compliant, data-preserving transformation approach that maintains both data and integrity constraints, and is also monotonic. We have evaluated S3PG using DBpedia 2022 and Bio2RDF CT. Our analysis confirmed S3PG's reliability and its ability to fully preserve data and integrity constraints in practice. The results also demonstrate that S3PG is more efficient in transforming RDF graphs into PGs than existing transformation approaches. Lastly, when transforming evolving graphs, S3PG preserves monotonicity and requires only a fraction of the time to incorporate updates. S3PG can effectively contribute to solving the interoperability issue between the RDF and PG data models for knowledge graphs. Open questions are how to extend the SHACL specification to the RDF-star and how to include it in S3PG's transformation algorithm, as well as how to create an automated query translator for S3PG. Moreover, support for ontological definitions and inference during transformations is also an important open challenge. Furthermore, the non-parsimonious transformation generates large PGs, an open question is how and when to optimize them. Finally, due to the reliance on shapes, errors or omissions in the shapes can result in erroneous transformations.

ACKNOWLEDGMENTS

This research was partially funded by the Danish Council for Independent Research (DFF) under grant agreement no. DFF-8048-00051B, the EU's H2020 research and innovation programme under grant agreement No 838216, and the Poul Due Jensen Foundation.

REFERENCES

[1] Renzo Angles, Angela Bonifati, Stefania Dumbrava, George Fletcher, Alastair Green, Jan Hidders, Bei Li, Leonid Libkin, Victor Marsault, Wim Martens, Filip Murlak, Stefan Plantikow, Ognjen Savkovic, Michael Schmidt, Juan Sequeda, Slawek Staworko, Dominik Tomaszuk, Hannes Voigt, Domagoj Vrgoc, Mingxi Wu, and Dusan Zivkovic. 2023. PG-Schema: Schemas for Property Graphs. *SIGMOD-2023* 1, 2 (2023), 198:1–198:25. https://doi.org/10.1145/3589778

[2] Renzo Angles, Angela Bonifati, Stefania Dumbrava, George Fletcher, Keith W Hare, Jan Hidders, Victor E Lee, Bei Li, Leonid Libkin, Wim Martens, et al. 2021. Pg-keys: Keys for property graphs. In *SIGMOD-2021*. 2423–2436.

[3] Renzo Angles, Aidan Hogan, Ora Lassila, Carlos Rojas, Daniel Schwabe, Pedro Szekely, and Domagoj Vrgoč. 2022. Multilayer graphs: A unified data model for graph databases. In *ACM SIGMOD Joint International Workshop: GRADES-NDA 2022*. 1–6.

[4] Renzo Angles, Harsh Thakkar, and Dominik Tomaszuk. 2019. RDF and Property Graphs Interoperability: Status and Issues. *AMW* 2369 (2019), 1–11.

[5] Renzo Angles, Harsh Thakkar, and Dominik Tomaszuk. 2020. Mapping RDF Databases to Property Graph Databases. *IEEE Access* 8 (2020), 86091–86110. https://doi.org/10.1109/access.2020.2993117

[6] Sören Auer, Christian Bizer, Georgi Kobilarov, Jens Lehmann, Richard Cyganiak, and Zachary G. Ives. 2007. DBpedia: A Nucleus for a Web of Open Data. In *ISWC-2007*, Vol. 4825. Springer, Busan, Korea, 722–735.

[7] Caleb Belth, Xinyi Zheng, Jilles Vreeken, and Danai Koutra. 2020. What is Normal, What is Strange, and What is Missing in an Knowledge Graph. In *The Web Conference*.

[8] Bio2rdf. 2020. Clinical Trials. https://download.bio2rdf.org/#/current/clinicaltrials/. Accessed on February 21, 2024.

[9] Iovka Boneva, Jérémie Dusart, Daniel Fernández-Álvarez, and José Emilio Labra Gayo. 2019. Shape Designer for ShEx and SHACL constraints. In *Proceedings of the ISWC 2019 Satellite Tracks (CEUR Workshop Proceedings)*, Vol. 2456. CEUR-WS.org, Auckland, New Zealand, 269–272.

[10] Angela Bonifati, Stefania Dumbrava, Emile Martinez, Fatemeh Ghasemi, Malo Jaffré, Pacôme Luton, and Thomas Pickles. 2022. DiscoPG: property graph schema discovery and exploration. *VLDB-2022* 15, 12 (2022), 3654–3657.

[11] Marco Brandizi, Ajit Singh, and Keywan Hassani-Pak. 2018. Getting the best of Linked Data and Property Graphs: rdf2neo and the KnetMiner use case.. In *SWAT4LS*.

[12] Hirokazu Chiba, Ryota Yamanaka, and Shota Matsumoto. 2020. G2GML: Graph to graph mapping language for bridging RDF and property graphs. In *ISWC-2020*. 160–175.

[13] Andrea Cimmino, Alba Fernández-Izquierdo, and Raúl García-Castro. 2020. Astrea: Automatic Generation of SHACL Shapes from Ontologies. In *ESWC*, Vol. 12123. Springer, Heraklion, Crete, Greece, 497.

[14] WWW Consortium. 2014. RDF 1.1. https://w3.org/RDF/. Accessed 10th April, 2024.

[15] WWW Consortium. 2014. RDF Schema 1.1. https://www.w3.org/TR/rdf-schema/. Accessed 10th April, 2024.

[16] Davide Di Pierro, Stefano Ferilli, and Domenico Redavid. 2023. LPG-Based Knowledge Graphs: A Survey, a Proposal and Current Trends. *Information* 14, 3 (2023), 154.

[17] Daniel Fernandez-Álvarez, Jose Emilio Labra-Gayo, and Daniel Gayo-Avello. 2022. Automatic extraction of shapes using sheXer. *Knowledge-Based Systems* 238 (2022), 107975.

[18] Nadime Francis, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Petra Selmer, and Andrés Taylor. 2018. Cypher: An evolving query language for property graphs. In *SIGMOD-2018*. 1433–1445.

[19] E. Haihong, Penghao Han, and Meina Song. 2020. Transforming RDF to Property Graph in Hugegraph. In *Proceedings of the 6th International Conference on Engineering MIS*. ACM. https://doi.org/10.1145/3410352.3410833

[20] Ihab F Ilyas, Theodoros Rekatsinas, Vishnu Konda, Jeffrey Pound, Xiaoguang Qi, and Mohamed Soliman. 2022. Saga: A platform for continuous construction and serving of knowledge at scale. In *SIGMOD-2022*. 2259–2272.

[21] Holger Knublauch and Dimitris Kontokostas. 2017. Shapes constraint language (SHACL). *W3C Candidate Recommendation* 11, 8 (2017).

[22] Ora Lassila, Michael Schmidt, Brad Bebee, Dave Bechberger, Willem Broekema, Ankesh Khandelwal, Kelvin Lawrence, Ronak Sharda, and Bryan Thompson. 2021. Graph? Yes! Which one? Help! *arXiv preprint arXiv:2110.13348* (2021).

[23] Maurizio Lenzerini. 2002. Data integration: A theoretical perspective. In *Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. 233–246.

[24] Matteo Lissandrini, Martin Brugnara, and Yannis Velegrakis. 2018. Beyond macrobenchmarks: microbenchmark-based graph database evaluation. *VLDB-2018* 12, 4 (2018), 390–403.

[25] Nandana Mihindukulasooriya, Mohammad Rifat Ahmmad Rashid, Giuseppe Rizzo, Raúl García-Castro, Óscar Corcho, and Marco Torchiano. 2018. RDF shape induction using knowledge base profiling. In *Proceedings of the 33rd Annual ACM Symposium on Applied Computing, SAC*. ACM, Pau, France, 1952–1959.

[26] Neo4j. 2023. Neo4j. https://neo4j.com. Accessed: April 1, 2024.

[27] Neo4j. 2023. neosemantics configuration. https://neo4j.com/labs/neosemantics/4.0/config/. Accessed: April 1, 2024.

[28] Neo4j. 2023. neosemantics (n10s): Neo4j RDF & Semantics toolkit. https://neo4j.com/labs/neosemantics/. Accessed: April 1, 2024.

[29] Natasha Noy, Yuqing Gao, Anshu Jain, Anant Narayanan, Alan Patterson, and Jamie Taylor. 2019. Industry-scale Knowledge Graphs: Lessons and Challenges: Five diverse technology companies show how it's done. *Queue* 17, 2 (2019), 48–75.

[30] Harshvardhan J. Pandit, Declan O'Sullivan, and Dave Lewis. 2018. Using Ontology Design Patterns To Define SHACL Shapes. In *Proceedings of the 9th Workshop on Ontology Design and Patterns (CEUR Workshop Proceedings)*, Vol. 2195. CEUR-WS.org, Monterey, USA, 67–71.

[31] Top Quadrant. 2023. TopBraid. https://www.topquadrant.com/products/topbraid-composer/. Accessed 10th April, 2024.

[32] Kashif Rabbani, Matteo Lissandrini, and Katja Hose. 2022. SHACL and ShEx in the Wild: A Community Survey on Validating Shapes Generation and Adoption. In *Proceedings of the ACM Web Conference 2022*. ACM, Online, Lyon, France, 260–263. https://www2022.thewebconf.org/PaperFiles/65.pdf

[33] Kashif Rabbani, Matteo Lissandrini, and Katja Hose. 2023. Extraction of Validating Shapes from very large Knowledge Graphs. *VLDB-2023* 16, 5 (2023), 1023–1032.

[34] rdf2pg. 2023. RDF2PG. https://github.com/renzoar/rdf2pg. Accessed: April 1, 2024.

[35] Ognjen Savkovic, Evgeny Kharlamov, and Steffen Lamparter. 2019. Validation of SHACL Constraints over KGs with OWL 2 QL Ontologies via Rewriting. In *ESWC-2019*, Vol. 11503. Springer, Slovenia, 314–329.

[36] Juan Sequeda and Ora Lassila. 2022. *Designing and building enterprise knowledge graphs*. Springer Nature.

[37] Juan F Sequeda, Marcelo Arenas, and Daniel P Miranker. 2012. On directly mapping relational databases to RDF and OWL. In *TheWebConf-2012*. 649–658.

[38] Thomas Pellissier Tanon, Gerhard Weikum, and Fabian M. Suchanek. 2020. YAGO 4: A Reason-able Knowledge Base. In *ESWC-2020*, Vol. 12123. Springer, Heraklion, Crete, Greece, 583–596.

[39] Dominik Tomaszuk, Renzo Angles, Łukasz Szeremeta, Karol Litman, and Diego Cisterna. 2019. Serialization for property graphs. In *Beyond Databases, Architectures and Structures. Paving the Road to Smart Data Processing and Analysis: In BDAS 2019, Ustroń, Poland, May 28–31, 2019, Proceedings 15*. Springer, 57–69.

[40] Denny Vrandečić and Markus Krötzsch. 2014. Wikidata: a free collaborative knowledgebase. *Commun. ACM* 57, 10 (2014), 78–85.

[41] Domagoj Vrgoč, Carlos Rojas, Renzo Angles, Marcelo Arenas, Diego Arroyuelo, Carlos Buil-Aranda, Aidan Hogan, Gonzalo Navarro, Cristian Riveros, and Juan Romero. 2023. MillenniumDB: An Open-Source Graph Database System. *Data Intelligence* 5, 3 (08 2023), 560–610. https://doi.org/10.1162/dint_a_00229 arXiv:https://direct.mit.edu/dint/article-pdf/5/3/560/2158194/dint_a_00229.pdf

[42] W3C. 2023. SHACL- core constraint components. https://www.w3.org/TR/shacl/#core-components. Accessed 10th April, 2024.

[43] W3C. 2023. W3C: RDF Type. http://www.w3.org/1999/02/22-rdf-syntax-ns#type. Accessed 10th April, 2024.

[44] Guanghui Zhang, Bo Yu, and Liping Bu. 2023. Bi-Mapping Between RDF and Property Graphs. In *ICTech-2023*. IEEE, 34–39.

[45] Ran Zhang, Pengkai Liu, Xiefan Guo, Sizhuo Li, and Xin Wang. 2019. A unified relational storage scheme for RDF and property graphs. In *Web Information Systems and Applications: In WISA 2019, Qingdao, China, September 20-22, 2019*. Springer, 418–429.