Bridging graph data models: RDF, RDF-star, and property graphs as directed acyclic graphs [extended abstract]

Ewout Gelling
Eindhoven University of Technology
ewoutgelling1999@gmail.com

George Fletcher
Eindhoven University of Technology
g.h.l.fletcher@tue.nl

Michael Schmidt
Amazon Web Services
schmdtm@amazon.com

ABSTRACT

Graph database users today face a choice between two technology stacks: the Resource Description Framework (RDF), on one side, is a data model with built-in semantics that was originally developed by the W3C to exchange interconnected data on the Web; on the other side, Labeled Property Graphs (LPGs) are geared towards efficient graph processing and have strong roots in developer and engineering communities. The two models look at graphs from different abstraction layers (triples in RDF vs. edges connecting vertices with inlined properties in LPGs), expose — at least at the surface — distinct features, come with different query languages, and are embedded into their own software ecosystems.

In this short paper, we introduce a novel unifying graph data model called *Statement Graphs*, which combines the traits of both RDF and LPG and achieves interoperability at different levels: it (a) provides the ability to manage RDF and LPG data as a single, interconnected graph, (b) supports querying over the integrated graph using any RDF or LPG query language, while (c) clearing the way for graph stack independent data exchange mechanisms and formats. We formalize our new model as directed acyclic graphs and sketch a system of bidirectional mappings between RDF, LPGs, and Statement Graphs. Our mappings implicitly define read query semantics for RDF and LPGs query languages over the unified data model, thus providing graph users with the flexibility to use the query language of their choice for their graph use cases.

As a proof of concept for our ideas, we also present the 1G Playground; an in-memory DBMS built on the concepts of Statement Graphs, which facilitates storage of both RDF and LPG data, and allows for cross-model querying using both SPARQL and Gremlin.

PVLDB Reference Format:
Ewout Gelling, George Fletcher, and Michael Schmidt. Bridging graph data models: RDF, RDF-star, and property graphs as directed acyclic graphs [extended abstract]. PVLDB, -(-): XXX-XXX, 2023.
doi:XX.XX/XXX.XX

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/aws-samples/amazon-neptune-samples/tree/master/1g-playground.

Proceedings of the VLDB Endowment, Vol. -, No. - ISSN XXXX-XXXX.
doi:XX.XX/XXX.XX

1 INTRODUCTION

The Resource Description Framework (RDF [32]) and Labeled Property Graphs (LPGs) are the two predominant graph data models encountered in today's industrial graph database landscape [24].

RDF has been developed and standardized as part of the W3C's Semantic Web [8] initiative. At its core, an RDF dataset is defined as a set of (*subject*, *predicate*, *object*) triples, where each triple represents a fact such as (*Alice*, *knows*, *Bob*). RDF triples can be grouped into containers called *named graphs*, and an RDF dataset is defined as a collection of such graphs. On top of RDF, the W3C has developed and standardized higher-level languages that support modeling and inference (RDFS [36], OWL [28]), as well as a declarative query language called SPARQL [31]. While not explicitly defined using graph concepts (such as vertices and edges), there is a close connection between RDF and graphs: every triple can be understood as an edge from a *subject* node under a given *predicate* label to an *object* node. In fact, the W3C itself labels RDF a "graph-based data model" [32] and leverages graphs as an intuitive way to visualize RDF (see e.g. [34]) – yet does not provide a formal mapping between RDF and common graph concepts such as nodes and edges.

In contrast, Labeled Property Graphs are formalized using graph terminology and concepts [2, 3, 10]. The LPG data model consists of sets of vertices and connecting edges, where both vertices and edges can be described through key-value pairs, so-called properties (e.g. [10]). In contrast to SPARQL, LPG query languages such as Gremlin [13] and openCypher [22] have dedicated constructs to access and operate over vertices and edges (edges are called "relationships" in openCypher), treating them as first-order concepts.

One specific feature that makes LPGs a prominent choice for real-world use cases is its built-in support for edge properties; for instance, the distance associated with a *route* edge connecting Frankfurt airport (FRA) with London Heathrow (LHR) can be attached to the edge route in form of a property *distance -> "655"*. RDF, where everything is a triple, has no built-in mechanism to attach properties to the "edge" (strictly speaking, there is not even a notion of an edge in RDF). The designated mechanism for modeling such scenarios in RDF is the reification vocabulary [36], which allows "statements about statements". In the example above, it can be used to express that a triple describing a route between two airports has a certain distance, by means of four additional triples:

```
(FRA, route, LHR), // original triple
(s, rdf:subject, FRA), (s, rdf:predicate, route),
(s, rdf:object, LHR), (s, distance, 655)
```

In this example, `s` is a new identifier that represents the underlying triple `(FRA, route, LHR)`; it is defined via pointers to the three position of the triple, using reserved predicates from the RDF namespace, `rdf:subject`, `rdf:predicate`, and `rdf:object`, respectively. The identifier `s` is then used in the last triple to attache

the distance. While conceptually sound, RDF reification has been criticized for its verbosity and poor usability [17, 23]. Other approaches to "model around" lacking edge property support in RDF have similiar limitations: n-ary relations [29], for instance, alter the graph topology and complicate querying; utilizing named graphs for reification (as proposed in [26]) occupies the graph container, which then can no longer be used for other purposes.

In response to the (usability) gap for edge properties in RDF, the W3C recently started the RDF-star working group[1], which aims to establish RDF extensions that provide a concise, user-friendly syntax for expressing edge properties in RDF (and to query them in SPARQL). While the working group seeks to address an important usability aspect of RDF, its outcome will (at best) close an existing gap in one of the standards – but not overcome the fundamental problem that RDF and LPG are two separate technology stacks that look at graphs from different layers of abstraction. Edge properties are only *one amongst many* examples where the stacks differ; they both have very unique strengths (and weaknesses). To give just a few examples, (i) RDF offers great support for global data exchange, publishing, and graph merging (e.g., [9]; (ii) LPG query languages, Gremlin and openCypher, offer first-level support for paths as data types, whereas SPARQL comes with built-in support for federation across different endpoints [30]; (iii) both RDF and LPGs come with their own software ecosystems, developer communities (e.g., Apache Tinkerpop [13]), and tooling around graph processing.

We argue that, in the end, graph users just want to solve their graph use cases. Having to choose between either of the two models – and, even worse, being locked into of the two stacks – stands in direct conflict with their desire to maintain flexibility in a world of changing requirements and emerging opportunities. In order to give these users the flexibility to address their graph problems in the most convenient way, in this paper we are exploring an approach to achieve graph data model interoperability through a novel, overarching data model, that has enough expressive power to capture the RDF, RDF-star, and property graphs data models alike. We next describe two concrete real-world use cases that illustrate the value of such an overarching graph data model.

Use Case: Foodie Travel. A new startup in the airline tickets industry wants to offer their customers the most cost effective way to travel to their destination. They have decided to use a property graph database system as their backing data store. Herein, airports are modeled as nodes, and flights between airports as edges. Each edge carries information about the flight's cost, in the form of an edge property. Use of openCypher path queries allows them to find the most cost-effective route to a customers destination, which may not always be a direct flight. Facing heavy competition, the startup decides to pivot, and cater to a more niche market; that of food enthusiasts. Their new value proposition is to provide the most economical flights to Michelin star rated restaurants. They know that the DBpedia [21] knowledge graph contains ample information about accredited restaurants, which they aim to integrate with their system. However, herein lies a problem: this data is only accessible as RDF. The inclusion of a triplestore in the architecture would result in additional query overhead and an increase in system complexity. Having both datasets in the same data system would be more practical and performant. Unfortunately, conversion of the RDF data into an LPG format is also ruled-out as an option, because this will make it more challenging to incorporate additional relevant RDF data in the future (for example, a filter for restaurants in buildings which are considered historic). In essence, the startup is in need of a way to combine both RDF and LPG data into a single, queryable structure, where original data is kept in-tact.

Use Case: Event Knowledge Graph Analytics. An *Event Knowledge Graph* (EKG) [12, 27] is an LPG based data model relevant to the field of process mining. It describes the logical flow of *events* in a business process. Events are modeled as graph nodes and carry a descriptive *activity name* such as "Create Invoice" or "Receive Payment". Transitions between events are modeled as directed edges, to which additional information can be linked in the form of *entities*. These are separate nodes in the graph, with unique identifiers. For example, the transition between the events "Receive payment" and "Clear invoice" might be associated with an entity containing information about that particular payment. This association is established by referencing the entity identifier in an edge property of a transition. An EKG can ultimately be used to discover the life cycle of a particular entity in the process, where event nodes along a path of transitions that reference said entity are aggregated with an openCypher path query.

Notice that the design of the EKG is affected by a mismatch between the features of the underlying data model and desired query language. The domain calls for the association of entity nodes with event transition edges, a functionality that is not inherently supported in the property graph model. This is solved by referencing the entity in an edge property of the transition. However, this indirect link can cause potential data integrity issues down the line. A more natural approach would be to model this domain in RDF-star, in which the entity can be directly linked with the transition through reification. However, this would complicate access to the model, since the SPARQL-star query language does not support the collection of nodes in arbitrary length path traversals. Again, in essence EKG is in need of a way to combine key features across the RDF and LPG stacks into a single queryable structure.

2 CONTRIBUTIONS

Towards addressing practical use cases such as these, commonly arising across application domains, in this short paper we make the following contributions:

- A formalization of the Statement Graph data model; an overarching data model for RDF, RDF-star, and property graphs.
- An open-source proof of concept implementation for Statement Graphs called the '1G Playground'[2]; an in-memory DBMS that supports both RDF and property graphs, and allows for cross-model querying in SPARQL and Gremlin.

Our contributions provide solid foundations and pave the way for graph stack independent management, querying, and exchange of graph data.

[1] https://www.w3.org/groups/wg/rdf-star

[2] https://github.com/aws-samples/amazon-neptune-samples/tree/master/1g-playground

3 BACKGROUND AND PROBLEM DESCRIPTION

There have been different efforts to define interoperability between the *surface* data models (RDF, RDF-star, and LPG) through direct mappings[1, 11, 16, 25]. These primarily make use of the fact that RDF(-star) triples and property graph edges are both 3-ary relations to convert information. However, because the models do not have fully compatible feature sets, these translation are inherently not lossless without introducing additional semantic meaning. Hence, we take a different approach to graph model interoperability: Instead of defining direct mappings between the data models, we raise the three to a common level of abstraction, which we use as an intermediate layer in our inter-data model mappings.

We start off by giving formal definitions for the three surface data models. Herein, we will refer to the terms *concrete types* and *concrete elements*. A *concrete type* acts as a building block of some arbitrary data model. Let $\mathcal{E}_1, \ldots, \mathcal{E}_k$ denote concrete types for some $k > 0$. Let an element $e \in \mathcal{E}_1, \ldots, \mathcal{E}_k$ be called a *concrete element*.

An example of a concrete type is the set of all property graph labels $\mathcal{K}$. In which the label $city \in \mathcal{K}$ is an example of a concrete element. We use the following 6 concrete types in our definitions: literals $\mathcal{L}$, IRI's $\mathcal{I}$, blank nodes $\mathcal{B}$, labels $\mathcal{K}$, property names $\mathcal{P}$, and property values $\mathcal{V}$.

An *RDF triple* is a tuple $t = (s, p, o)$ where $s \in \mathcal{I} \cup \mathcal{B}$ is called the *subject*, $p \in \mathcal{I}$ is called the *predicate*, and $o \in \mathcal{I} \cup \mathcal{B} \cup \mathcal{L}$ is called the *object*. An *RDF graph* is a finite set consisting of zero or more RDF triples. An *RDF data set* is a collection of the form $\{G_D, (g_i, G_i), \cdots, (g_n, G_n)\}$, for which $0 \leq i \leq n$, where G_D and G_i are both RDF graphs, and $g_i \in \mathcal{I} \cup \mathcal{B}$. The graph G_D is referred to as the *default graph*, and the pairs of form (g_i, G_i) are referred as *named graphs*[34]. Table 1 contains an example of an RDF data set in which a DBpedia[21] fragment of a (fictional) restaurant is modeled.

Table 1: An RDF data set in Turtle[35] syntax containing data about a restaurant.

RDF - (default graph)

```
@prefix ex: <http://example.org/> .
@prefix dbr: <http://dbpedia.org/resource/> .
@prefix dbp: <http://dbpedia.org/property/> .
@prefix dbo: <http://dbpedia.org/ontology/> .

ex:ChezSG
          a dbr:Restaurant ;
          dbo:cuisine "French, classical"@en ;
          dbp:rating "Michelin guide"@en ;
          dbp:city dbr:Paris .
dbr:Paris
          a dbp:city ;
          dbp:name "Paris"@en .
```

An *RDF-star triple* is a tuple $t = (s, p, o)$ where $s \in \mathcal{I} \cup \mathcal{B} \cup \{t_1\}$, $p \in \mathcal{I}$, and $o \in \mathcal{I} \cup \mathcal{B} \cup \mathcal{L} \cup \{t_2\}$, in which t_1 and t_2 are RDF-star triples (adapted from [17]). Reference cycles are not allowed in an RDF-star triple, i.e., $t \neq t_1, t_2$ nor can t ever be referenced in t_1 or t_2 recursively. The definitions of an *RDF-star graph* and *RDF-star data set* are then constructed in similar fashion to their RDF counterparts.

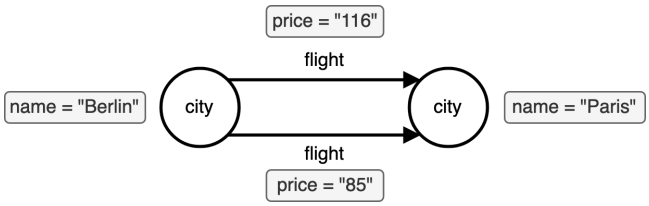

Figure 1: A property graph of flight data.

The working definition of a property graph is adapted from Tomaszuk et al. [25]. It is generalized and inclusive, and allows both nodes and edges to be associated with a variable number of labels and properties. A property graph is a tuple: $G = (N, A, P, \delta, \lambda, \sigma, \rho)$ where:

(1) N is a finite set of nodes, A is a finite set of edges, P is a finite set of properties, and N, A, P are mutually disjoint.
(2) $\delta : A \rightarrow (N \times N)$ is a total function that associates each edge in A with a pair of nodes in N;
(3) $\lambda : (N \cup A) \rightarrow 2^{\mathcal{K}}$ is a total function that associates nodes and edges to a set of labels (possibly empty);
(4) $\sigma : (N \cup A) \rightarrow 2^{P}$ is a total function that associates nodes and edges to a set of properties (possibly empty), satisfying that $\sigma(o_1) \cap \sigma(o_2) = \emptyset$ for each pair $o_1, o_2 \in dom(\sigma)$;
(5) $\rho : P \rightarrow (\mathcal{P} \times \mathcal{V})$ is a total function that assigns a property name-value pair to each property.

Figure 1 shows an example of a property graph in which two flights between Berlin and Paris are modeled as edges.

4 STATEMENT GRAPHS

The Statement Graph data model must be sufficiently flexible to capture the unique traits of all three surface data models simultaneously. This is achieved by making Statement Graphs a semantically agnostic data model, that is to say, by not placing any constraints on data semantics. We formalize statement graphs as directed acyclic graphs (Definition 4.1), in which each internal node has exactly three outgoing edges. These edges have fixed labels, which indicates the ordering of the vertices they point to. Each leaf node of the Statement Graph is associated with a concrete element, and each internal node with a so-called statement identifier (from the concrete type of statement identifiers, $\mathcal{S}$). An internal vertex and its out-neighbours then essentially make up a piece of identifiable information, which we consequently refer to as a *statement*. Finally, internal nodes can have direct connections to other internal nodes, on the condition that their connecting edge is not labeled *predicate*. Note that we do not imply any sort of physical implementation strategy here, Statement Graphs are a purely logical and conceptual model. Graph users do not interact directly with Statement Graphs, instead, they get the convenience of using the surface data model of their choice.

Definition 4.1. A *Statement Graph* G is a directed vertex- and edge-labeled graph $G = (V, E, \tau)$, where:

- V is a finite set of vertices;
- $E \subseteq V \times \{subject, predicate, object\} \times V$ is a finite set of labeled edges;
- $\tau : V \rightarrow \mathcal{S} \cup \mathcal{E}_1 \cup \ldots \cup \mathcal{E}_k$ is a total injective function that labels vertices with statement identifiers or other concrete elements;
- every vertex $v \in V$ satisfies the following conditions:
  (1) If $\tau(v) \in \mathcal{E}_1 \cup \cdots \cup \mathcal{E}_k$ then:
    (a) v has no outgoing edges,
    (b) v has at least 1 incoming edge.
  (2) If $\tau(v) \in \mathcal{S}$ then v has exactly 3 outgoing edges:
    (a) $(v, subject, v_s)$ and $\tau(v_s) \in \mathcal{S} \cup \mathcal{E}_1 \cup \cdots \cup \mathcal{E}_k$,
    (b) $(v, predicate, v_p)$ and $\tau(v_p) \in \mathcal{E}_1 \cup \cdots \cup \mathcal{E}_k$,
    (c) $(v, object, v_o)$ and $\tau(v_o) \in \mathcal{S} \cup \mathcal{E}_1 \cup \cdots \cup \mathcal{E}_k$.
- G is acyclic, i.e., there is no path in E from a vertex to itself.

Figures 2 and 3 each show an example of a Statement Graph. In these figures, *subject* edges are represented as hollow arrows, *predicate* edges as double arrows, and *object* edges as single arrows. Note that we could have taken an alternative approach to formalizing Statement Graphs, e.g., as a class of nested hypergraphs [19]. We adopt the view of directed acyclic graphs, for ease of presentation.

5 CONCRETE STATEMENT GRAPHS

In this section, we introduce concise Statement Graph fragments for our surface data models, which we refer to as Statement Graph *Images*. These are designed to be minimal, i.e., each Image captures the traits of its associated surface data model, and nothing more. Lossless, bidirectional mappings exist between the Statement Graph Images and their corresponding surface data models, which shows that they are functionally equivalent. This system of mappings has been omitted for brevity, but is included in the Appendix of the full version of this extended abstract[15].

5.1 RDF data

In the following definitions we refer to the notion of vertex *isomorphism* within Statement Graphs. We consider two internal vertices to be isomorphic if they have outgoing edges to the same vertices; they are essentially making the "same" statement. This allows RDF-(star) set semantics to be expressed in Statement Graphs.

The traits of RDF-star are captured in an *RDF-star-Image* (Definition 5.1) by distinguishing between two groups of statements: triple statements and membership statements. A triple statement captures the semantics of an RDF-star triple, reification is herein accommodated by allowing statement identifiers to be referenced in *subject* and *object* position (see constraints 1, 2, and 3). A membership statement assigns a triple statement to either a graph name, or the default graph identifier, $G_D \in \mathcal{I}$. The constant concrete element $in \in \mathcal{I}$ is used in *predicate* position of these statements. The final constraint (4), first lays out the structure of a membership statement (4a), then asserts that each triple statement is referenced in the context of at least one other statement (4b), and ensures statement uniqueness by disallowing vertex *isomorphism* (4c).

Definition 5.1. A Statement Graph is an *RDF-star-Image* when:

(1) For all edges $(v, subject, v_s) \in E$, $\tau(v_s) \in \mathcal{S} \cup \mathcal{I} \cup \mathcal{B}$;
(2) For all edges $(v, predicate, v_p) \in E$, $\tau(v_p) \in \mathcal{I}$;
(3) For all edges $(v, object, v_o) \in E$, $\tau(v_o) \in \mathcal{S} \cup \mathcal{I} \cup \mathcal{B} \cup \mathcal{L}$;
(4) For every internal vertex $v \in V$, let v_s, v_p, v_o be vertices with incoming *subject*, *predicate*, *object* edges from v, the following constraints hold:
  (a) if $\tau(v_p) = in$ then $\tau(v_s) \in \mathcal{S}$ and $\tau(v_o) \in \mathcal{I} \cup \mathcal{B}$ and v has no incoming edges;
  (b) if $\tau(v_p) \in \mathcal{I} \setminus in$ then v has at least 1 incoming edge;
  (c) There is no $v' \in V$ that is *isomorphic* to v.

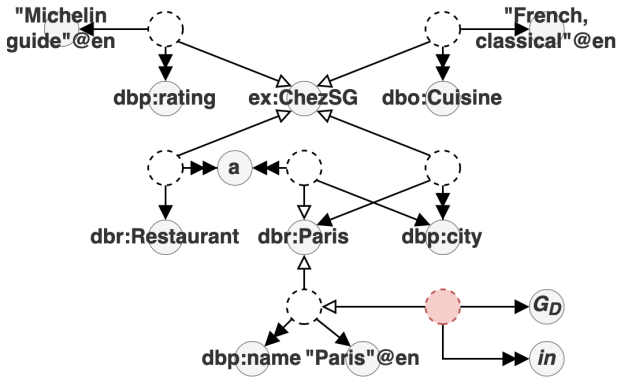

Figure 2: Statement Graph of restaurant data. This is the RDF-Image of the RDF data set given in Table 1.

An *RDF-Image* (Definition 5.2) is a special case of RDF-star-Image in which internal nodes can only be referenced in the context of a membership statement. Consequently, each RDF-Image must satisfy two additional constraints on top of those that it inherits, these are: (1) Internal nodes can never reference others in *object* position, and (2) every internal node must have at least one incoming edge from a membership statement (i.e. be part of at least one graph). Figure 2 shows a Statement Graph that is the RDF-Image of the data set from Table 1. In this figure, the node marked in red is a membership statement, all other membership statements are omitted to avoid cluttering.

Definition 5.2. A Statement Graph is an *RDF-Image* when it is an RDF-star-Image and:

(1) For all edges $(v, object, v_o) \in E$, $\tau(v_o) \in \mathcal{I} \cup \mathcal{B} \cup \mathcal{L}$;
(2) For every internal vertex $v \in V$, let v_s, v_p, v_o be vertices with incoming *subject*, *predicate*, *object* edges from v, the following constraint holds:
  (a) if $\tau(v_p) \in \mathcal{I} \setminus in$ then $\tau(v_s) \in \mathcal{I} \cup \mathcal{B}$ and $\tau(v_o) \in \mathcal{I} \cup \mathcal{B} \cup \mathcal{L}$ and v has at least 1 incoming edge.

5.2 LPG data

Let $\mathcal{N}$ be the concrete type of property graph node identities. For simplicity, we assume that each property graph node is associated with a concrete element from this set. The traits of a property graph are captured in an *LPG-Image* (Definition 5.3) by distinguishing between three groups of statements: Edge statements (4a), node

label statements (4b), and property statements (4c). The constant concrete element $l_p \in 2^{\mathcal{K}}$ is used in *predicate* position of node label statements. Figure 3 shows a Statement Graph that is the LPG-Image of the property graph from Figure 1. In this figure, $n1$ and $n2$ are elements from $\mathcal{N}$.

Definition 5.3. A Statement Graph is an *LPG-Image* when:

(1) For all edges $(v, subject, v_s) \in E$, $\tau(v_s) \in \mathcal{N} \cup \mathcal{S}$;
(2) For all edges $(v, predicate, v_p) \in E$, $\tau(v_p) \in \mathcal{P} \cup 2^{\mathcal{K}}$;
(3) For all edges $(v, object, v_o) \in E$, $\tau(v_o) \in \mathcal{V} \cup \mathcal{N} \cup 2^{\mathcal{K}}$;
(4) For every internal vertex $v \in V$, let v_s, v_p, v_o be vertices with incoming $subject$, $predicate$, $object$ edges from v, precisely one of the following constraints hold:
   (a) if $\tau(v_p) \in 2^{\mathcal{K}} \setminus l_p$ then $\tau(v_s) \in \mathcal{N}$ and $\tau(v_o) \in \mathcal{N}$;
   (b) if $\tau(v_p) = l_p$ then $\tau(v_s) \in \mathcal{N}$ and $\tau(v_o) \in 2^{\mathcal{K}}$ and v has no incoming edges;
   (c) if $\tau(v_p) \in \mathcal{P}$ then $\tau(v_s) \in \mathcal{S} \cup \mathcal{N}$ and $\tau(v_o) \in \mathcal{V}$ and v has no incoming edges.

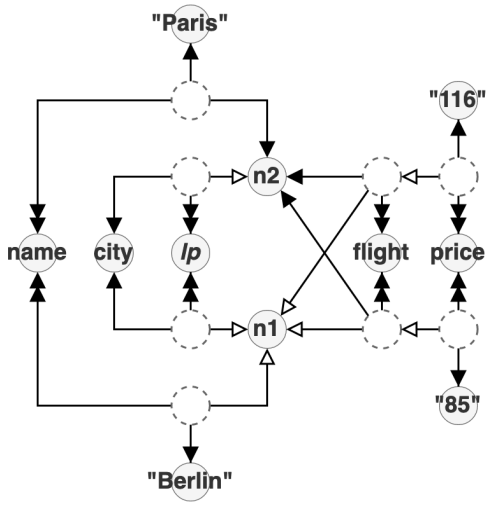

Figure 3: Statement Graph of flight data. This is the LPG-Image of the property graph from Figure 1

5.3 OneGraph data

A *OneGraph-Image* (Definition 5.4) is a Statement Graph comprised of a combination of RDF-, RDF-star-, and LPG-Images. It is used in the next section to simplify reasoning about interoperability. The name 'OneGraph-Image' is a reference to the OneGraph (1G) vision of Lassila et al. [20]. As shown in [14], our formalization of Statement Graphs, and more specifically that of OneGraph-Images, acts as a concrete implementation of the OneGraph vision.

Definition 5.4. A Statement Graph G is a *OneGraph-Image* when each component of G is an RDF-star-Image or LPG-Image, disregarding constraints on isomorphism of vertices.

6 INTEROPERABILITY

We have shown that Statement Graphs act as a common abstraction layer for the surface data models, which allows their instances to be aggregated under a single structure. In this section, we focus on how Statement Graphs are able to facilitate their interoperability. We convey this through a series of mappings that show how a OneGraph-Image can be converted into an RDF-, RDF-star-, or LPG-Image. These mappings implicitly define cross-model read query semantics for the surface data models. For example, an RDF Data set can be queried in Gremlin by subsequently converting it into an RDF-Image, LPG-Image, and property graph.

The mappings make use of the following total functions that relate naturally similar concrete types from the RDF and property graph worlds. They are: $literalvalue : \mathcal{L} \mapsto \mathcal{V}$, $blank2node : \mathcal{B} \mapsto \mathcal{N}$, $IRI2node : \mathcal{I} \mapsto \mathcal{N}$, $IRI2prop : \mathcal{I} \mapsto \mathcal{P}$, and $IRI2label : \mathcal{I} \mapsto 2^{\mathcal{K}}$. There is also an inverse to each of these that achieves the opposite goal, e.g., $value2literal : \mathcal{V} \mapsto \mathcal{L}$, etc. These functions are used to convert the concrete elements associated with leaf nodes of the given Statement Graph. Their entries act as parameters to the mappings. For example, an entry in *literal2value*, relating the literal `"Paris"@en` with property value `"Paris"`.

In the below translations, vertex removals are recursive, i.e., when a vertex is removed, all vertices with outgoing edges to that vertex are also removed.

6.1 OneGraph-Image to RDF-(star)-Image

The following mapping translates the OneGraph-Image $G = (V, E, \tau)$ into an RDF-Image if the optional step (5) is taken, or an RDF-star-Image if the optional step is skipped.

(1) For each leaf node $v \in V$, convert $\tau(v)$ using the appropriate type conversion function if needed, merge nodes with equivalent values for τ;
(2) Add vertices v_{in} and v_{G_D} to V, with $\tau(v_{in}) = in$ and $\tau(v_{G_D}) = G_D$, if such vertices did not already exist in V;
(3) For every internal vertex $v \in V$, add vertex v_{st} to V, $\tau(v_{st}) \in \mathcal{S}$, add edges edges $(v_g, subject, v)$, $(v_g, predicate, in)$, and $(v_g, object, v_{G_D})$ to E;
(4) For every internal vertex $v \in V$, remove vertices in V that are isomorphic to v;
(5) (Optional) For every internal vertex $v \in V$, let v_s, v_p, v_o be vertices with incoming $subject$, $predicate$, $object$ edges from v. If either v_s or v_o is an internal vertex and $\tau(v_p) \neq in$, then remove v;
(6) Remove vertices in V without incoming or outgoing edges.

The RDF-Image of the RDF data set of Table 1 obtained via this mapping is visualized in Figure 2.

6.2 OneGraph-Image to LPG-Image

The following mapping translates the OneGraph-Image $G = (V, E, \tau)$ into an LPG-Image.

(1) For each leaf node $v \in V$, convert $\tau(v)$ using the appropriate type conversion function if needed, merge nodes with equivalent values for τ. When encountering a $\tau(v) \in \mathcal{I}$:
   (a) if v has an incoming $subject$ or $object$ edge and no incoming $predicate$ edge; Set $\tau(v) = IRI2node(\tau(v))$
   (b) if v has an incoming $predicate$ edge and no incoming $subject$ or $object$ edge; Set $\tau(v) = IRI2label(\tau(v))$
   (c) if v has an incoming $subject$ or $object$ edge and an incoming $predicate$ edge;

(i) Add node $\overline{v}$ to V, let $\tau(\overline{v}) = IRI2node(\tau(v))$;
(ii) Let $\tau(v) = IRI2label(\tau(v))$;
(iii) Let all $predicate$ edges point to v, let all $subject$ and $object$ edges point to $\overline{v}$.

(2) For each internal vertex $v \in V$, let v_s, v_p, v_o be vertices with incoming $subject$, $predicate$, $object$ edges from v:
   (a) If v has an incoming edge from some vertex $\overline{v}$, then remove $\overline{v}$ if one of the following conditions hold:
      (i) The incoming edge to v is not labeled $subject$
      (ii) $\tau(v_o) \notin \mathcal{N}$
      (iii) $\overline{v}$ does not have an $object$ edge to a vertex $\overline{v_o}$ with $\tau(\overline{v_o}) \in \mathcal{V}$.
(3) Remove vertices in V without incoming or outgoing edges.

The LPG-Image of the property graph of Figure 1 obtained via this mapping is visualized in Figure 3.

7 1G PLAYGROUND

We next describe our open source implementation of Statement Graphs, which we call the "1G Playground", highlighting the OneGraph vision which inspired this work. The open source 1G Playground[3] complements the foundational work in this paper with a practical, demonstrable product that illustrates the key ideas of Statement Graphs and serves as a proof of concept. It consists of two components that adhere to the traditional client-server model. On the client side we have a REPL environment akin to psql[4], while the server side acts as a simple in-memory DBMS (see Figure 4). The server exposes a REST interface with endpoints for the following functionalities: data loading, data retrieving, querying, and modification of current settings. Data can be loaded and retrieved in various different RDF and property graph formats, such as: N-Quads[33], Turtle[35], Graphson[5], and many more. The Playground also comes with a novel shared serialization format that allows users to export their RDF and LPG data in a unified '1G' syntax (accessible using flag *-og*).

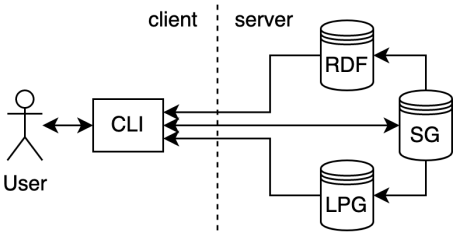

Figure 4: Architecture of the 1G Playground

The server component of the playground is made up of a single OneGraph-Image (indicated by SG in the figure), which is accompanied by two derivative data stores; one for LPG (backed by TinkerPop) and one for RDF (backed by RDF4J[6]). The user can not modify these data stores directly, only the Statement Graph itself can be written to. RDF and LPG input first gets converted into its corresponding Image before being added to the central graph. The translations described in Section 6 are then used to reload both derivative data stores from the updated Statement Graph. The playground allows the user to configure the $IRI2label$, $label2IRI$, $IRI2prop$ and $prop2IRI$ functions, which gives them a certain level of control over these translations. All other concrete type conversions (such as $literal2value$) are performed implicitly. This architecture allows the playground to evaluate SPARQL and Gremlin read queries on the Statement Graph, without having to implement a custom query engine from scratch.

The playground includes a set of interactive demo scenarios which we have developed to help users understand the tool, and to illustrate Statement Graphs and their capabilities. There are three scenarios that can be explored using the *scenario* command, more info about this can be found on the GitHub page.

8 RELATED WORK

Mapping between RDF, RDF-star, and labeled property graphs. In recent years there has been quite some effort on designing direct model-to-model mappings between RDF, RDF-star, and LPGs, e.g., [1, 11, 16, 25]. Our work differs from and contributes to this body of work by providing an overarching data model supporting uniform translations between all of these surface data models. Furthermore, taking the data model perspective provides a more fundamental understanding of what graphs are and what is common and different between graphs modeled as RDF, RDF-star, and LPG (and beyond).

Unifying (graph) data models. The study of graph modeling (e.g., [4, 10, 18]) and unifying "meta" data models (e.g., [6, 7]) are classic themes in data management research. Most closely related to our investigation are the OneGraph [20] and Multilayer Graph [5] proposals, which also take a statement-centric approach to unifying RDF, RDF-star, and LPG. Our approach complements this work by taking a novel view of statement graphs as directed acyclic graphs, which are familiar, intuitive, and easy to manipulate and reason about.

9 CONCLUSION AND FUTURE WORK

We have argued that graph users just want to solve their graph use cases, and having to bridge multiple graph data models and graph stacks, as is often the case in contemporary graph work, stands in direct conflict with this desire. In this paper we have developed solid foundations on which to design and engineer graph systems which resolve this conflict by bridging the worlds of RDF and property graphs, with Statement Graphs, an intuitive and simple unifying graph data model. We have also described our open source toolkit, 1G Playground, which demonstrates the feasibility and usefulness of Statement Graphs.

Many interesting research challenges arise from our work. A main focus here can be development and study of the broader framework around Statement Graph-based systems, e.g., query languages and query rewriting, schema and constraint languages and enforcement methods, indexing strategies, and serialization formats. A concrete way forward on many of these lines of future work could be through further development of the open source 1G Playground.

[3] https://github.com/aws-samples/amazon-neptune-samples/tree/master/1g-playground
[4] https://www.postgresql.org/docs/current/app-psql.html
[5] https://tinkerpop.apache.org/docs/3.4.1/dev/io/#graphson
[6] https://rdf4j.org/

REFERENCES

[1] Ghadeer Abuoda, Daniele Dell'Aglio, Arthur Keen, and Katja Hose. 2022. Transforming RDF-star to Property Graphs: A Preliminary Analysis of Transformation Approaches – extended version. In *QuWeDa Workshop on Storing, Querying and Benchmarking Knowledge Graphs*.

[2] Renzo Angles, Angela Bonifati, Stefania Dumbrava, George Fletcher, Alastair Green, Jan Hidders, Bei Li, Leonid Libkin, Victor Marsault, Wim Martens, Filip Murlak, Stefan Plantikow, Ognjen 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. In *SIGMOD*.

[3] Renzo Angles, Angela Bonifati, Stefania Dumbrava, George Fletcher, Keith W. Hare, Jan Hidders, Victor E. Lee, Bei Li, Leonid Libkin, Wim Martens, Filip Murlak, Josh Perryman, Ognjen Savkovic, Michael Schmidt, Juan F. Sequeda, Slawek Staworko, and Dominik Tomaszuk. 2021. PG-Keys: Keys for Property Graphs. In *SIGMOD*. 2423–2436.

[4] Renzo Angles and Claudio Gutierrez. 2008. Survey of Graph Database Models. *ACM Comput. Surv.* 40, 1, Article 1 (2008), 39 pages.

[5] 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 *Proceedings of the 5th ACM SIGMOD Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA)*. Association for Computing Machinery, Article 11, 6 pages.

[6] Paolo Atzeni, Giorgio Gianforme, and Paolo Cappellari. 2009. A Universal Metamodel and Its Dictionary. *Trans. Large Scale Data Knowl. Centered Syst.* 1 (2009), 38–62.

[7] Paolo Atzeni and Riccardo Torlone. 1993. A metamodel approach for the management of multiple models and translation of schemes. *Inf. Syst.* 18, 6 (1993), 349–362.

[8] Tim Berners-Lee, James Hendler, and Ora Lassila. 2001. The Semantic Web. *Scientific american* 284, 5 (2001), 34–43.

[9] Christian Bizer, Tom Heath, Kingsley Idehen, and Tim Berners-Lee. 2008. Linked data on the web (LDOW2008). In *Proceedings of the 17th international conference on World Wide Web*. 1265–1266.

[10] Angela Bonifati, George Fletcher, Hannes Voigt, and Nikolay Yakovets. 2018. *Querying graphs*. Morgan & Claypool Publishers.

[11] Julian Bruyat, Pierre-Antoine Champin, Lionel Médini, and Frédérique Laforest. 2021. PREC: semantic translation of property graphs. *1st workshop on Squaring the Circles on Graphs* (2021). https://doi.org/arXiv:2110.12996v1

[12] Stefan Esser and Dirk Fahland. 2021. Multi-Dimensional Event Data in Graph Databases. Journal on Data Semantics, 109–141. https://doi.org/10.1007/s13740-021-00122-1

[13] The Apache Software Foundation. [n.d.]. Apache Tinkerpop: Gremlin Query Language. Retrieved Feb 26, 2023 from https://tinkerpop.apache.org/gremlin.html

[14] Ewout Gelling. 2022. *Bridging graph data models: RDF, RDF-Star, and property graphs as directed acyclic graphs*. Master's thesis. Eindhoven University of Technology.

[15] Ewout Gelling, George Fletcher, and Michael Schmidt. 2023. Bridging graph data models: RDF, RDF-star, and property graphs as directed acyclic graphs [extended abstract]. https://github.com/EwoutGelling/Bridging-data-models

[16] Olaf Hartig. 2014. Reconciliation of RDF* and Property Graphs. *arXiv:1406.3399v3* (2014). https://doi.org/10.48550/ARXIV.1409.3288

[17] Olaf Hartig and Bryan Thompson. 2014. Foundations of an alternative approach to reification in RDF. *arXiv preprint arXiv:1406.3399* (2014).

[18] Borislav Iordanov. 2010. HyperGraphDB: A Generalized Graph Database. *Web-Age Information Management* (2010), 25–36.

[19] Cliff Joslyn and Kathleen Nowak. 2017. Ubergraphs: A Definition of a Recursive Hypergraph Structure. *arXiv:1704.05547v1* (2017). https://doi.org/10.48550/ARXIV.1704.05547

[20] O. Lassila, M. Schmidt, B. Bebee, D. Bechberger, W. Broekema, A. Khandelwal, K. Lawrence, R. Sharda, and B. Thompson. 2023. The OneGraph Vision: Challenges of Breaking the Graph Model Lock-In. *Semantic Web Journal* 14 (2023). Issue 1.

[21] Pablo N Mendes, Max Jakob, and Christian Bizer. 2012. DBpedia: A multilingual cross-domain knowledge base. In *Proceedings of the Eighth International Conference on Language Resources and Evaluation (LREC'12)*. European Language Resources Association (ELRA), 1813–1817. http://www.lrec-conf.org/proceedings/lrec2012/pdf/570_Paper.pdf

[22] Inc. Neo4j. [n.d.]. openCypher. Retrieved Feb 26, 2023 from http://opencypher.org/

[23] Vinh Nguyen, Olivier Bodenreider, and Amit Sheth. 2014. Don't like RDF reification? Making statements about statements using singleton property. In *Proceedings of the 23rd international conference on World wide web*. 759–770.

[24] Yuanyuan Tian. 2023. The World of Graph Databases from An Industry Perspective. *ACM SIGMOD Record* 51, 4 (2023), 60–67.

[25] Dominik Tomaszuk, Renzo Angles, and Harsh Thakkar. 2020. PGO: Describing Property Graphs in RDF. *IEEE Access* 8 (2020), 118355–118369. https://doi.org/10.1109/access.2020.3002018

[26] Johannes Trame, Carsten Keßler, and Werner Kuhn. 2013. Linked data and time–modeling researcher life lines by events. In *Spatial Information Theory: 11th International Conference, COSIT 2013, Scarborough, UK, September 2-6, 2013. Proceedings 11*. Springer, 205–223.

[27] W.M.P. van der Aalst and J. Carmona. 2022. Process Mining Handbook. (2022), 274–320.

[28] W3C. 2004. OWL Web Ontology Language. https://www.w3.org/TR/owl-ref/

[29] W3C. 2006. Defining N-ary Relations on the Semantic Web (W3C Working Group Note. https://www.w3.org/TR/swbp-n-aryRelations/

[30] W3C. 2013. SPARQL 1.1 Federated Query. https://www.w3.org/TR/sparql11-federated-query/

[31] W3C. 2013. SPARQL 1.1 Query Language. https://www.w3.org/TR/sparql11-query/

[32] W3C. 2014. RDF 1.1 Concepts and Abstract Syntax. https://www.w3.org/TR/rdf11-concepts/

[33] W3C. 2014. RDF 1.1 N-Quads. https://www.w3.org/TR/n-quads/

[34] W3C. 2014. RDF 1.1 Primer. https://www.w3.org/TR/rdf11-primer/

[35] W3C. 2014. RDF 1.1 Turtle. https://www.w3.org/TR/turtle/

[36] W3C. 2014. RDF Schema 1.1. https://www.w3.org/TR/rdf-schema/

10 APPENDIX

10.1 Property graph to Statement Graph

The following mapping can be applied to property graph $PG = (N, A, P, \delta, \lambda, \sigma, \rho)$ to obtain an LPG-Image $G = (V, E, \tau)$

(1) For every $n \in N$:
  (a) Add vertices v_s, v_p, v_o, and v_{st} to V, let $\tau(v_s) = n$, $\tau(v_p) = l_p$, $\tau(v_o) = \lambda(n)$ and let $\tau(v_{st})$ be an unused element from $\mathcal{S}$
  (b) Add edges $(v_{st}, subject, v_s)$, $(v_{st}, predicate, v_p)$, and $(v_{st}, object, v_o)$ to E.
  (c) For every property $(p_{name}, p_{value}) \in \rho(\sigma(n))$:
    (i) Add vertices $\overline{v_p}$, $\overline{v_o}$ and $\overline{v_{st}}$ to V, let $\tau(\overline{v_p}) = l_p$, $\tau(\overline{v_o}) = \lambda(n)$ and let $\tau(\overline{v_{st}})$ be an unused element from $\mathcal{S}$
    (ii) Add edges $(\overline{v_{st}}, subject, v_s)$, $(\overline{v_{st}}, predicate, \overline{v_p})$, $(\overline{v_{st}}, object, \overline{v_o})$ to E.
(2) For every $e \in A$, where $\delta(e) = (n_1, n_2)$:
  (a) Add vertices v_s, v_p, v_o, and v_{st} to V, let $\tau(v_s) = n_1$, $\tau(v_p) = \lambda(e)$, $\tau(v_o) = n_2$ and let $\tau(v_{st})$ be an unused element from $\mathcal{S}$
  (b) Add edges $(v_{st}, subject, v_s)$, $(v_{st}, predicate, v_p)$, and $(v_{st}, object, v_o)$ to E.
  (c) For every property $(p_{name}, p_{value}) \in \rho(\sigma(e))$:
    (i) Add vertices $\overline{v_p}$, $\overline{v_o}$ and $\overline{v_{st}}$ to V, let $\tau(\overline{v_p}) = l_p$, $\tau(\overline{v_o}) = \lambda(n)$ and let $\tau(\overline{v_{st}})$ be an unused element from $\mathcal{S}$
    (ii) Add edges $(\overline{v_{st}}, subject, v_{st})$, $(\overline{v_{st}}, predicate, \overline{v_p})$, $(\overline{v_{st}}, object, \overline{v_o})$ to E.
(3) Merge vertices in V with equal values for τ;

10.2 Statement Graph to Property graph

The following mapping can be applied to LPG-Image $G = (V, E, \tau)$ to obtain a property graph $PG = (N, A, P, \delta, \lambda, \sigma, \rho)$.

(1) Let V_{st} be the set of internal vertices in V. When iterating V_{st}, let v_s, v_p, v_o be vertices with incoming $subject$, $predicate$, $object$ edges from a $v \in V_{st}$
(2) For every $v \in V_{st}$ for which $\tau(v_p) = l_p$:
  (a) Let node $n = \tau(v_s)$, add n to N, set $\lambda(n) = \tau(v_s)$.
(3) For every $v \in V_{st}$ for which $\tau(v_s) \in \mathcal{N} \wedge \tau(v_o) \in \mathcal{N}$:
  (a) Let node $n_1 = \tau(v_s)$ and $n_2 = \tau(v_o)$, add edge $e = (n_1, n_2)$ to A, set $\lambda(e) = \tau(v_p)$.
(4) For every $v \in V_{st}$ for which $\tau(v_o) \in \mathcal{V}$:
  (a) If $\tau(v_s) \in \mathcal{N}$, let node $n = \tau(v_s)$, add property p to P and entries $\sigma(n) = p$ and $\rho(p) = (\tau(v_p), \tau(v_o))$
  (b) If $\tau(v_s) \in \mathcal{S}$, obtain edge e that was previously created for v_s, add property p to P and entries $\sigma(e) = p$ and $\rho(p) = (\tau(v_p), \tau(v_o))$

10.3 RDF data set to Statement Graph

The following mapping can be applied to RDF data set $D = \{G_D, (g_i, G_i), \cdots, (g_n, G_n)\}$ to obtain an RDF-Image $G = (V, E, \tau)$

(1) Add v_{in} to V, let $\tau(v_{in}) = in$
(2) For all elements $g \in D$:
  (a) Let $(name, graph) = (g_i, G_i)$ if g is a named graph, or let $(name, graph) = (G_{default}, G_D)$ if g is the default graph;
  (b) Add a new vertex v_{graph} to G, let $\tau(v_{graph}) = name$;
  (c) For all RDF triples $(s, p, o) \in graph$:
    (i) Add v_s to G, let $\tau(v_s) = s$;
    (ii) Add v_p to G, let $\tau(v_p) = p$;
    (iii) Add v_o to G, let $\tau(v_o) = o$;
    (iv) Add v_{st} to G, let $\tau(v_{st})$ be an unused element from $\mathcal{S}$
    (v) Add edges $(v_{st}, subject, v_s)$, $(v_{st}, predicate, v_p)$ and $(v_{st}, object, v_o)$ to E;
    (vi) Add v_{mem} to G, let $\tau(v_{mem})$ be an unused element from $\mathcal{S}$
    (vii) Add edges $(v_{mem}, subject, v_{st})$, $(v_{mem}, predicate, v_{in})$ and $(v_{mem}, object, v_{graph})$ to E;
(3) Merge all isomorphic vertices;
(4) Merge vertices in V with equal values for τ.

10.4 Statement Graph to RDF data set

The following mapping can be applied to RDF-Image $G = (V, E, \tau)$ to obtain an RDF data set $D = \{G_D, (g_i, G_i), \cdots, (g_n, G_n)\}$, let D contain only the default graph.

(1) Let V_{st} be the set of internal vertices in V. When iterating V_{st}, let v_s, v_p, v_o be vertices with incoming $subject$, $predicate$, $object$ edges from a $v \in V_{st}$;
(2) For every $v \in V_{st}$ for which $\tau(v_p) \neq in$:
  (a) Create RDF triple $t = (s, p, o)$;
  (b) Let $s = \tau(v_s)$, $p = \tau(v_p)$, $o = \tau(v_o)$;
  (c) Let V_{mem} be the set of vertices with outgoing edges to v;
  (d) For all $v_{mem} \in V_{mem}$, let $\overline{v_o}$ be the vertex with incoming $object$ edge from v_{mem}:
    (i) Let $name = \tau(\overline{v_o})$;
    (ii) If $name = G_D$, add the triple t to the default graph in D.
    (iii) If $name \neq G_D$, add the triple t to the named graph named $name$ in D, if no such graph exists create a new one.

10.5 RDF-star data set to Statement Graph

The following mapping can be applied to RDF-star data set $D = \{G_D, (g_i, G_i), \cdots, (g_n, G_n)\}$ to obtain an RDF-star-Image $G = (V, E, \tau)$

(1) Add v_{in} to V, let $\tau(v_{in}) = in$
(2) For all elements $g \in D$:
  (a) Let $(name, graph) = (g_i, G_i)$ if g is a named graph, or let $(name, graph) = (G_{default}, G_D)$ if g is the default graph;
  (b) Add a new vertex v_{graph} to G, let $\tau(v_{graph}) = name$;
  (c) For all RDF-star triples $(s, p, o) \in graph$:
    (i) If s is an RDF-star triple, then execute steps (i), (ii), (iii), (iv), and (v) on t. Let v_s be the internal vertex obtained in step (v);
    (ii) Else, add v_s to G, let $\tau(v_s) = s$;
    (iii) Add v_p to G, let $\tau(v_p) = p$;

(iv) If o is an RDF-star triple, then execute steps (i), (ii), (iii), (iv), and (v) on t. Let v_o be the internal vertex obtained in step (v);
(v) Else, add v_o to G, let $\tau(v_o) = o$;
(vi) Add v_{st} to G, let $\tau(v_{st})$ be an unused element from $\mathcal{S}$;
(vii) Add edges $(v_{st}, subject, v_s)$, $(v_{st}, predicate, v_p)$ and $(v_{st}, object, v_o)$ to E;
(viii) Add v_{mem} to G, let $\tau(v_{mem})$ be an unused element from $\mathcal{S}$;
(ix) Add edges $(v_{mem}, subject, v_{st})$, $(v_{mem}, predicate, v_{in})$ and $(v_{mem}, object, v_{graph})$ to E;

(3) Merge all isomorphic vertices;
(4) Merge vertices in V with equal values for τ.

10.6 Statement Graph to RDF-star data set

The following mapping can be applied to RDF-star-Image $G = (V, E, \tau)$ to obtain an RDF-star data set $D = \{G_D, (g_i, G_i), \cdots, (g_n, G_n)\}$, let D contain only the default graph.

(1) Let V_{st} be the set of internal vertices in V. When iterating V_{st}, let v_s, v_p, v_o be vertices with incoming $subject$, $predicate$, $object$ edges from a $v \in V_{st}$;
(2) For every $v \in V_{st}$ for which $\tau(v_p) \neq in$:
(a) Create RDF-star triple $t = (s, p, o)$;
(b) Let $s = \tau(v_s)$, if $s \in S$ then execute steps (a), (b), (c), and (d) on v_s. Let s be the RDF-star triple obtained in step (d);
(c) Let $p = \tau(v_p)$;
(d) Let $o = \tau(v_o)$, if $o \in S$ then execute steps (a), (b), (c), and (d) on v_o. Let o be the RDF-star triple obtained in step (d);
(e) Let V_{mem} be the set of vertices with outgoing edges to v;
(f) For all $v_{mem} \in V_{mem}$, let $\overline{v_o}$ be the vertex with incoming $object$ edge from v_{mem}:
(i) Let $name = \tau(\overline{v_o})$;
(ii) If $name = G_D$, add the triple t to the default graph in D;
(iii) If $name \neq G_D$, add the triple t to the named graph named $name$ in D, if no such graph exists create a new one.