Translating RDF 1.2 Graphs to Property Graphs

Maxime Jakubowski[1,*], Dominik Tomaszuk[2], Daniel Fernández-Álvarez[3], Ruben Taelman[4], Ruben Dedecker[4], Jose-Emilio Labra-Gayo[3] and Katja Hose[1]

[1]TU Wien, Austria
[2]University of Bialystok, Poland
[3]University of Oviedo, Spain
[4]Department of Electronics and Information Systems, Ghent University – imec, Belgium

Abstract

RDF 1.2 introduces statement-level constructs, such as triple terms, reifiers, and descriptions, that pose new challenges for transforming RDF data into labeled property graphs (LPGs). This work presents and compares three algorithms for translating RDF 1.2 data into LPGs, each interpreting a different amount of the RDF 1.2 reification vocabulary. Algorithm Structural interprets no vocabulary and losslessly supports arbitrary RDF 1.2 graphs by representing triple terms explicitly as nodes; Direct interprets `rdf:reifies` to recover annotated edges while preserving reifiers as graph objects; and Fold additionally folds reifier descriptions into edge properties, resulting in a more concise, albeit lossy, LPG representation. We define the three translations, analyze which class of RDF 1.2 graphs each preserves, and experimentally evaluate their implementations. The experiments confirm the predicted trade-offs: all variants scale linearly on the tested workloads, the generality of Structural has no runtime penalty, and folding descriptions into edge records can significantly reduce the number of edges on description-heavy data.

Keywords

RDF, Property Graphs, Data Modeling

1. Introduction

Graph-structured data has become an increasingly important means of representing and integrating complex information across a wide range of application domains. As graph-based applications continue to grow in scale and complexity, so does the need to exchange data between different graph data management systems and data models.

RDF and labeled property graphs (LPGs) are the two predominant models for graph-structured data. Although both represent information as nodes connected by edges, they differ in how they model relationships and their associated metadata. In many applications, edges/relationships carry additional metadata, such as provenance, confidence scores, timestamps, or sources. LPGs support this natively, since edges may themselves have properties. RDF, by contrast, has no native mechanism for attaching metadata directly to a triple, motivating a long line of work on statement-level metadata, most prominently RDF-star [1].

RDF 1.2, now at Candidate Recommendation status [2], standardizes a new mechanism for capturing such meta. Its central construct is the *triple term*, an RDF triple used as a term in the object position of another triple. Metadata is attached through a *reifier*, a resource linked to a triple term by the dedicated predicate `rdf:reifies`. Interoperability with property graphs was an explicit motivation for the W3C RDF & SPARQL Working Group chartered to standardize RDF-star [3]. At first sight, the correspondence appears straightforward: the edge-with-properties construct that LPGs provide natively has a natural counterpart in RDF through the combined use of reifiers and triple terms.

DMKG'26: 2nd International Workshop on Data Management for Knowledge Graphs, October 2026, Bari, Italy
*Corresponding author.
maxime.jakubowski@tuwien.ac.at (M. Jakubowski); fernandezalvdaniel@uniovi.es (D. Fernández-Álvarez)
0000-0002-7420-1337 (M. Jakubowski); 0000-0003-1806-067X (D. Tomaszuk); 0000-0002-8666-7660 (D. Fernández-Álvarez); 0000-0001-5118-256X (R. Taelman); 0000-0002-3257-3394 (R. Dedecker); 0000-0001-8907-5348 (J. Labra-Gayo); 0000-0001-7025-8099 (K. Hose)
© 2026 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

CEUR Workshop Proceedings ceur-ws.org ISSN 1613-0073
published 2026-11-09

CEUR-WS.org/Vol-4322/paper_18.pdf

```
:sun :madeOf :plasma ~ :r2 .

:r1 rdf:reifies <<( :sun :madeOf :popcorn )>> ;
    :statedBy :alice ;
    :certainty 0.8 .

:r3 rdf:reifies <<( :sun :madeOf :popcorn )>> ;
    :statedBy :bob .
```

Figure 1: An RDF 1.2 graph in Turtle. The first statement is asserted and carries the reifier `:r2`; the proposition that the sun is made of popcorn is *not* asserted, and is reified twice.

The correspondence is nonetheless less direct than it first appears, as RDF 1.2 permits configurations that an LPG cannot represent. A triple term may be the object of any predicate, not only `rdf:reifies`; it may contain another triple term, nesting to arbitrary depth; and the triple it represents need not itself be asserted, that is, included as a statement in the RDF graph. An LPG edge is more rigid: it cannot be the endpoint of another edge, and it cannot be nested. The apparent correspondence between an annotated triple and an edge with properties is therefore a modeling pattern that RDF 1.2 data may follow, not one guaranteed by the data model. Translating RDF 1.2 into LPGs consequently requires deciding how to handle graphs that depart from this pattern.

Figure 1 illustrates three challenges that arise from this mismatch.

Referenceability. The triple `:r1 :statedBy :alice` states that `:sun :madeOf :popcorn` was stated by Alice. Ideally, this annotation should be stored in an edge's record. However, record entries are plain values, whereas `:alice` denotes a graph object in its own right (and might be an endpoint of other edges). Descriptions thus relate statements to graph objects, while edge properties hold values; a translation must bridge this gap, for instance by references from records to nodes.

Assertedness. The first statement of Figure 1 is written with `~:r2`, a shorthand that both asserts the triple (i.e., it also includes it as a statement of the graph) and reifies it. In contrast, `rdf:reifies` alone simply refers to a triple term without asserting it: the triple reified by `:r1` and `:r3` is not asserted. Whether a triple term is asserted is thus independent of its reification. An LPG edge has no such freedom; it is simply present. So, a translation must resolve the distinction one way or another: record assertedness explicitly, encode it structurally, or discard it.

Multiplicity. Both `:r1` and `:r3` reify the same triple term. More generally, RDF 1.2 allows several reifiers to relate to the same triple term, and a single reifier to relate to several triple terms. The correspondence between reifiers and triple terms is many-to-many. Then, interpreting every reifier as a distinct LPG edge is a modeling assumption rather than a consequence of the RDF 1.2 data model.

These challenges expose a central design choice: *how much of the RDF 1.2 reification vocabulary should a translation interpret?* At one extreme, a translation interprets none of it and represents every triple term as a node: the complete graph is preserved, but the LPG serves as a generic container in which no annotation is ever stored as an edge property. At the other extreme, a translation interprets `rdf:reifies` and folds annotations into edge records, where an LPG natively keeps them, but, at the price of restricting the inputs and discarding information that does not fit.

We investigate this design space through three translations. Structural interprets no vocabulary and losslessly supports arbitrary RDF 1.2 graphs by representing triple terms explicitly. Direct interprets `rdf:reifies`, representing each reified triple as an edge while preserving reifiers as graph objects. Fold additionally folds reifier descriptions into edge records (which is the native LPG representation of annotated statements) at the cost of reifier identity. We formally define the translations, analyze their trade-offs, and evaluate them experimentally.

Translation between RDF and property graphs has been studied extensively in both directions. Angles et al. [4] define three direct RDF-to-property-graph mappings, two of which preserve both information and semantics, while Rabbani et al. [5] additionally exploit SHACL and PG-Schema during translation. Hartig [6] formalizes the property graph model and defines transformations in both directions; for the

opposite direction specifically, Das et al. [7] represent property graphs as RDF in Oracle, including a reification-based encoding. Closest to our setting, Abuoda et al. [8] analyze transformations from RDF-star to property graphs and identify the cases in which existing approaches fail. Our work differs in targeting RDF 1.2. Previous approaches consider RDF 1.1 or RDF-star, where quoted triples correspond more directly to annotated LPG edges. RDF 1.2 replaces that design with reifiers, restricts triple terms to object position, and gives `rdf:reifies` a generic interpretation.

The remainder of this paper is organized as follows. Section 2 introduces the required notation. Section 3 presents the translation approaches, each illustrated using the running example of Figure 1. Section 4 evaluates the approaches, and Section 5 concludes this paper with an outlook to future work.

2. Preliminaries

We briefly recall the two data models involved: RDF 1.2 graphs and labeled property graphs.

RDF 1.2 Graphs Let IRIs, Blanks, and Literals be mutually disjoint, countably infinite sets of *IRIs*, *blank nodes*, and *literals*; these are the *basic RDF terms*. To account for RDF 1.2's nested triples, the set of *RDF terms* Terms and the set of *RDF triples* Triples are defined by mutual recursion. An RDF triple (s, p, o) has a *subject* $s \in \mathsf{IRIs} \cup \mathsf{Blanks}$, a *predicate* $p \in \mathsf{IRIs}$, and an *object* $o \in \mathsf{Terms}$; that is, $\mathsf{Triples} \subseteq (\mathsf{IRIs} \cup \mathsf{Blanks}) \times \mathsf{IRIs} \times \mathsf{Terms}$. An RDF term is an IRI, a blank node, a literal, or a triple: $\mathsf{Terms} = \mathsf{IRIs} \cup \mathsf{Blanks} \cup \mathsf{Literals} \cup \mathsf{Triples}$. An *RDF graph* G is a finite set of RDF triples, $G \subseteq \mathsf{Triples}$. A triple that is an element of G is said to be *asserted* in G, sometimes also called a *top-level* triple.

Triple Terms and Reification An RDF triple that occurs as the object of another triple is called a *triple term*. The same triple may occur as an asserted triple, as a (nested) triple term, or both; the two are independent. Indeed, a nested triple need not be asserted. RDF 1.2 uses the IRI $\mathit{reifies} \in \mathsf{IRIs}$[1] in the predicate of so-called *reifing triples*. A *reifying triple* is a triple $(r, \mathit{reifies}, \tau)$ whose object τ is a triple term and whose subject r is called the *reifier* of τ [2]. Reifier r is used in further triples that describe τ. We call a triple (r, p, o) with $p \neq \mathit{reifies}$, whose subject r is a reifier of τ, a *description triple* of τ.

Labeled Property Graphs Let Nodes and Edges be mutually disjoint, countably infinite sets representing *Nodes* and *Edges*, respectively. Let Keys be a countably infinite set of *property keys*, and let Values be a countably infinite set of *property values*. The set of all *Labels* is denoted by Labels, and the set of all *Records* (finite mappings from keys to values) is defined as $\mathsf{Records} = \{r : K \to \mathsf{Values} \mid K \subseteq \mathsf{Keys}\}$. A *labeled property graph* G is defined as a tuple $G = (N, E, \mathsf{edg}, \mathsf{lab}, \mathsf{rec})$ where:

- $N \subseteq \mathsf{Nodes}$ is a finite set of nodes; $E \subseteq \mathsf{Edges}$ is a finite set of edges such that $N \cap E = \emptyset$;
- $\mathsf{edg} : E \to (N \times N)$ is a total function mapping each edge to an ordered pair of nodes, where for $e \in E$ with $\mathsf{edg}(e) = (u, v)$, we call u the *source* and v the *target* of e;
- $\mathsf{lab} : (N \cup E) \to 2^{\mathsf{Labels}}$ is a total function mapping nodes and edges to finite sets of labels (including the empty set);
- $\mathsf{rec} : (N \cup E) \to \mathsf{Records}$ is a total function mapping nodes and edges to records (their *properties*).

We say a node or edge o *carries* a property key k when k is defined in $\mathsf{rec}(o)$, and that it *carries* $k \mapsto v$ when additionally $\mathsf{rec}(o)(k) = v$.

3. Translating RDF 1.2 to LPGs

The three translations share the same principle: every basic RDF term occurring as a subject or object becomes a node (literals included, so a literal is represented once as a node shared by all triples mentioning it, rather than inlined as a value) and every triple becomes an edge, labeled by its predicate.

[1] The full *reifies* IRI is http://www.w3.org/1999/02/22-rdf-syntax-ns#reifies

Throughout, blank nodes are Skolemized and treated as IRIs. The translations deviate from this only in their treatment of triple terms, reifiers, descriptions, and assertedness (cf. Table 1). STRUCTURAL interprets no vocabulary. DIRECT and FOLD interpret *reifies*, representing each reified triple as an edge: DIRECT keeps reifiers as nodes referenced from these edges, while FOLD folds a reifier's descriptions into the edge record itself, the native LPG representation of an annotated statement.

From RDF terms to LPG nodes We reconcile the two data models with a single naming convention. Writing Strings for the set of all finite strings, we assume Strings $\subseteq$ Nodes $\cap$ Values $\cap$ Labels, so that one string may serve as a node, as a property value, and as a label. Fix an injective function ID : Terms $\to$ Strings that assigns every RDF term a distinct string identifier[2]. Since Strings $\subseteq$ Nodes, the identifier $\text{ID}(t)$ is itself a node, and we take it as the node representing the RDF term t. Indeed, injectivity is what makes node identity sound: two occurrences of the same term yield the same node.

Our translations reference nodes from within edge records. However, concrete systems keep node identifiers internal and do not expose them as addressable values [9]. To keep the translations realizable, every node additionally carries its identifier as an ordinary property under the reserved key refId: the node of t carries refId $\mapsto \text{ID}(t)$, and edges never carry refId. A reference is then an equality between property values, in the manner of a *foreign key*: a referring key, carried on an edge, holds the refId of the node it denotes. Each variant has its own referring keys: in for STRUCTURAL, reifiedBy for DIRECT, and, for FOLDit is references.

Predicates, in contrast, become labels: the predicate p of a triple becomes the label $\text{ID}(p) \in$ Labels of its edge (an IRI occurring both as a predicate and as a subject or object thus results in both a label and a node). Finally, two helpers, shared by all algorithms, type and populate the node of a term t. KIND : Terms $\to$ Labels returns IRI, Literal, or TripleTerm according to whether $t \in$ IRIs, $t \in$ Literals, or $t \in$ Triples. RECORD : Terms $\to$ Records returns $\{\text{refId} \mapsto \text{ID}(t)\}$, extended for literals with value and datatype (plus language or direction for language-tagged and directional strings).

3.1. Translation STRUCTURAL

STRUCTURAL interprets no RDF 1.2 vocabulary; in particular, it treats *reifies* as an ordinary predicate. Each triple term τ is materialized as a node $\text{ID}(\tau)$ labeled "TripleTerm". Being a node, $\text{ID}(\tau)$ can be pointed at by edges, so any triple whose object is τ (e.g., a reifying triple $(r, \mathit{reifies}, \tau)$ or otherwise) is just a plain edge into $\text{ID}(\tau)$, with no special handling. Then, to represent the triple $\tau = (s, p, o)$ itself, we emit it as an edge e from s to o labeled p. However, e must be distinguished from an assertion and it must be tied back to $\text{ID}(\tau)$. Since we cannot simply create an edge from e to $\text{ID}(\tau)$ itself (that would mean making e an endpoint of another edge, which LPGs forbid), STRUCTURAL uses the refId introduced above. The edge key in $\in$ Keys is added to the record of e which carries the reference to $\text{ID}(\tau)$ Furthermore, an edge with in $\mapsto \text{ID}(\tau)$ as part of its record represents the triple term τ, whereas an edge with *no* in key represents an asserted triple of G. Assertedness is thus recorded structurally; one can verify assertedness by checking the presence of in.

Throughout this section we illustrate each translation on the running example of Figure 1. We build it up gradually: we begin with the small fragment that already exercises the essential case for STRUCTURAL, then add the asserted triple, the second description, and the second reifier as they become relevant for DIRECT and FOLD.

Example 1. *We start from the smallest interesting fragment of Figure 1. Let* $\tau_1 = (\mathit{sun}, \mathit{madeOf}, \mathit{popcorn})$ *and* $G_{\text{STRUCTURAL}} = \{(r_1, \mathit{reifies}, \tau_1), (r_1, \mathit{statedBy}, \mathit{alice})\}$. *Here* r_1 *reifies* τ_1*, and the description* $(r_1, \mathit{statedBy}, \mathit{alice})$ *records who stated it. Note that* τ_1 *is* not *asserted: it occurs only as a triple term, never as a top-level triple of* $G_{\text{STRUCTURAL}}$*.* STRUCTURAL *creates five nodes:* $\text{ID}(\mathit{sun})$*,* $\text{ID}(r_1)$*,* $\text{ID}(\mathit{alice})$*, and* $\text{ID}(\mathit{popcorn})$ *labeled* IRI*, and* $\text{ID}(\tau_1)$ *labeled* TripleTerm*, and three edges: two for the triples of* $G_{\text{STRUCTURAL}}$*, and one carrying* in $\mapsto \text{ID}(\tau_1)$ *for the content of* τ_1*. Figure 2 shows the output* $\text{STRUCTURAL}(G_{\text{STRUCTURAL}})$.

[2] An example of such a function is the N-Triples serialization.

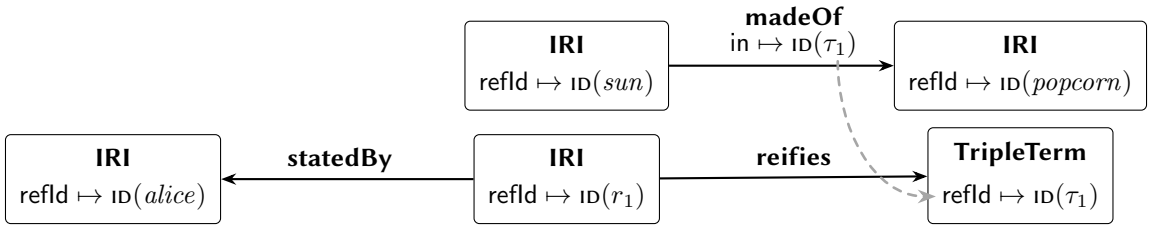


Figure 2: The STRUCTURAL encoding of Example 1. Solid arrows are LPG edges. The gray dashed arrow indicates the foreign key reference.

Algorithm 1 has a shared state consisting of the five LPG components and an additional set *encoded* across all procedure calls. The core procedure EMIT turns one triple into one edge; its second argument records the context of that triple, either $\perp$ (an asserted triple, top level) or the identifier of the enclosing triple term. When the object is itself a triple term τ that has not yet been emitted, EMIT recurses on τ to emit its triple; deeper nesting is handled by the same recursion. The guard on line 12 emits each triple term exactly once, so a triple term with several reifiers keeps its distinct *reifies* edges (one per reifier) while sharing a single node with label 'TripleTerm' and a single copy of its triple.

Algorithm 1 STRUCTURAL: interprets no vocabulary

Shared state: N, E, edg, lab, rec, *encoded*

1: **procedure** STRUCTURAL(G)
2: $N \leftarrow \emptyset$; $E \leftarrow \emptyset$; edg, lab, rec $\leftarrow$ empty maps; *encoded* $\leftarrow \emptyset$
3: **for all** $(s, p, o) \in G$ **do**
4: EMIT$\big((s, p, o),\ \perp\big)$ ▷ $\perp$: this is a graph triple
5: **return** $(N, E,$ edg, lab, rec$)$

6: **procedure** EMIT($(s, p, o),\ in$) ▷ $in \in$ Values $\cup \{\perp\}$
7: ADDNODE(s); ADDNODE(o)
8: choose $e \in$ Edges $\setminus E$; $E \leftarrow E \cup \{e\}$
9: edg$(e) \leftarrow \big($ID$(s),\ ID(o)\big)$
10: lab$(e) \leftarrow \{$ID$(p)\}$ ▷ the predicate is the edge label
11: rec$(e) \leftarrow \begin{cases} \emptyset & \text{if } in = \perp \\ \{\text{in} \mapsto in\} & \text{otherwise} \end{cases}$
12: **if** $o \in$ Triples **and** ID$(o) \notin$ *encoded* **then** ▷ triple term whose triple is not yet emitted
13: *encoded* $\leftarrow$ *encoded* $\cup \{$ID$(o)\}$
14: EMIT$\big(o,\ ID(o)\big)$ ▷ emit its triple; nesting handled by o's object

15: **procedure** ADDNODE(t) ▷ materialize t as a node
16: **if** ID$(t) \notin N$ **then**
17: $N \leftarrow N \cup \{$ID$(t)\}$
18: lab(ID(t)) $\leftarrow \{$KIND$(t)\}$
19: rec(ID(t)) $\leftarrow$ RECORD(t)

3.2. Translation DIRECT

DIRECT makes two assumptions about its input: (i) a triple term occurs only as the object of a *reifies* triple; (ii) triple terms are not nested. We call an RDF 1.2 graph satisfying both a *flat (reification) graph*. The assumptions are imposed by the structure of LPG edges: an edge cannot be the endpoint of another edge, nor can it contain one, so a triple term that is the object of an arbitrary predicate, or that nests another triple term, cannot be represented as an edge at all. In flat graphs, every triple term can be represented as an edge: it is an RDF term using basic RDF terms, occuring only in reifying triples. The excluded nesting structures are exactly those that STRUCTURAL handles by turning triple terms into

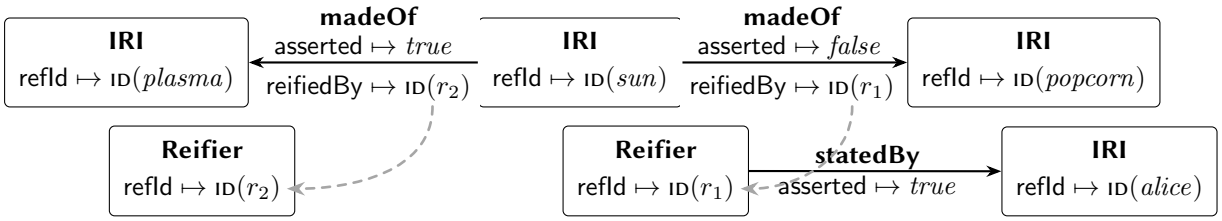


Figure 3: The DIRECT encoding of Example 2.

nodes; DIRECT targets flat reification graphs and leaves the rest to STRUCTURAL.

DIRECT interprets *reifies*: where STRUCTURAL keeps a reifying triple $(r, \mathit{reifies}, \tau)$ as an edge into a TripleTerm node, DIRECT represents the triple term $\tau = (s, p, o)$ itself as an edge from s to o labeled p, which we call a *triple term edge*. Such an edge is distinguished from an ordinary one by carrying reifiedBy $\mapsto$ ID(r), a foreign key to the node of its reifier r (recall that ID(r) carries refId). The reifier r always becomes a node labeled Reifier, serving as the target of this foreign key.

Example 2. *We extend $G_{\text{STRUCTURAL}}$ of Example 1 with an asserted triple and its reifier. Let $\tau_2 = (\mathit{sun}, \mathit{madeOf}, \mathit{plasma})$ and $G_{\text{DIRECT}} = G_{\text{STRUCTURAL}} \cup \{(\mathit{sun}, \mathit{madeOf}, \mathit{plasma}), (r_2, \mathit{reifies}, \tau_2)\}$. In contrast to τ_1, the triple τ_2 is* asserted*: it occurs both as the triple term reified by r_2 and as a plain triple of G_{DIRECT}. Unlike STRUCTURAL, DIRECT never turns a reifying triple into an edge of its own; each reifying triple instead becomes a single edge recording its reifier and its assertion status. Here this creates six nodes: ID(sun), ID$(\mathit{popcorn})$, ID(plasma), and ID(alice) labeled* IRI*, and ID(r_1), ID(r_2) labeled* Reifier*; and three edges: the edge for τ_1 with* asserted $\mapsto$ `false`*; the edge for τ_2, created for r_2 and then flipped to* asserted $\mapsto$ `true` *by the plain triple $(\mathit{sun}, \mathit{madeOf}, \mathit{plasma})$; and the asserted edge $(r_1, \mathit{statedBy}, \mathit{alice})$ for the description. Each of the first two edges references its reifier node, r_1 and r_2 respectively. Figure 3 shows the output DIRECT(G_{DIRECT}).*

When a triple term edge is created, it is not yet known whether it is asserted, since the triple τ may also occur as a plain triple of G. Algorithm 2 therefore proceeds in two passes. Pass 1 creates, for each reifying triple, one triple term edge carrying asserted $\mapsto$ `false`, and records it in the map *ttEdges*.[3] Pass 2 considers each top-level triple (s, p, o). If *ttEdges* already holds edges for (s, p, o), then that triple is asserted after all and every such edge is flipped to asserted $\mapsto$ `true`; otherwise (s, p, o) is an ordinary triple and becomes a new edge carrying asserted $\mapsto$ `true`. RDF 1.2 treats the occurrence of a triple as a triple term or top-level triple independently. Representing both occurrences using a single LPG edge carrying the asserted property is therefore a design choice of the translation.

3.3. Translation FOLD

FOLD considers flat reification graphs (cf. Section 3.2) satisfying two further assumptions: reifiers occur in subject position only, and each reifier reifies a single triple term (several reifiers per triple term remain allowed); we call such graphs *strict* flat (reification) graphs. Without the latter assumption, Pass 1 below would fold a reifier's description onto every triple term it reifies. Like DIRECT, FOLD interprets *reifies* and represents a reified triple $\tau = (s, p, o)$ as a triple term edge from s to o. It differs in what it does with the reifier. While DIRECT turns a reifier into a node, FOLD never makes a reifier a node. Instead, descriptions for τ (i.e., triples where the subject is a reifier cf. Section 2) are stored in the record of the triple term edges. We call the process of storing descriptions in edge records "folding." A reifier therefore is not a node of its own, and descriptions are folded into the triple term edge for τ.

Two folding strategies are offered. *Fixed folding* uses two reserved keys: refProperty for the description predicate, and references for a foreign key to the description object. As these keys are fixed, one edge holds a single description, so a reifier with several descriptions requires one edge per description. *Open folding* instead uses each description predicate itself as a property key, with the object's foreign key as

[3] Note that a triple term can have multiple reifiers. Here, this results in parallel triple term edges (one per reifying triple).

Algorithm 2 DIRECT: reifies-interpreting translation

Shared state: $N, E, \mathsf{edg}, \mathsf{lab}, \mathsf{rec}$

procedure DIRECT(G)
  $N \leftarrow \emptyset$; $E \leftarrow \emptyset$; edg, lab, rec $\leftarrow$ empty maps
  ttEdges $\leftarrow$ empty map ▷ maps triples to triple term edges
  Pass 1: each reifying triple creates one triple term edge
  for all $(r, \mathit{reifies}, \tau) \in G$ **with** $\tau \in \mathsf{Triples}$ **do**
    AddReifier(r)
    $e \leftarrow$ CreateEdge$\big(\tau,\ \{\mathsf{asserted} \mapsto \mathtt{false},\ \mathsf{reifiedBy} \mapsto \text{ID}(r)\}\big)$
    ttEdges$(\tau) \leftarrow$ *ttEdges*$(\tau) \cup \{e\}$
  Pass 2: plain triples assert triple term edges or create new edges
  for all $(s, p, o) \in G$ **with** $o \notin \mathsf{Triples}$ **do**
    if *ttEdges*$(s, p, o) \neq \emptyset$ **then** ▷ (s, p, o) is a triple term edge
      for all $e \in$ *ttEdges*(s, p, o) **do**
        rec(e)(asserted) $\leftarrow$ `true` ▷ triple term edge asserted iff its triple is in G
    else
      CreateEdge$\big((s, p, o),\ \{\mathsf{asserted} \mapsto \mathtt{true}\}\big)$
  return $(N, E, \mathsf{edg}, \mathsf{lab}, \mathsf{rec})$

procedure CreateEdge((s, p, o), *rec*) ▷ one edge carrying record *rec*
  AddNode(s); AddNode(o) ▷ AddNode as in Algorithm 1
  choose $e \in \mathsf{Edges} \setminus E$; $E \leftarrow E \cup \{e\}$
  edg$(e) \leftarrow \big(\text{ID}(s),\ \text{ID}(o)\big)$; lab$(e) \leftarrow \{\text{ID}(p)\}$
  rec$(e) \leftarrow$ *rec*
  return e

procedure AddReifier(r) ▷ materialize a reifier as a node
  if ID$(r) \notin N$ **then**
    $N \leftarrow N \cup \{\text{ID}(r)\}$
    lab(ID(r)) $\leftarrow \{\mathsf{Reifier}\}$
    rec(ID(r)) $\leftarrow$ Record(r) ▷ carries refId $\mapsto$ ID(r)

its record value, so that all descriptions of a reifier fit in a single triple term edge. This is more concise, but requires two restrictions: predicate identifiers must serve as property keys (Strings $\subseteq$ Keys), so the record has no predetermined key set; and since a record maps each key to one value, no two descriptions of the same reifier may share a predicate.[4] In both strategies the description object is represented as a node, addressable through the foreign key in the triple term edge.

Example 3. *We extend $G_{\textsc{Direct}}$ of Example 2 with the second description of r_1 and a second reifier r_3 of τ_1, completing the graph of Figure 1: $G_{\textsc{Fold}} = G_{\textsc{Direct}} \cup \{(r_1, \mathit{certainty}, 0.8),\ (r_3, \mathit{reifies}, \tau_1),\ (r_3, \mathit{statedBy}, \mathit{bob})\}$.*

Figure 4 shows Fold($G_{\textsc{Fold}}$) under open folding. None of the three description triples becomes an edge, and no reifier becomes a node: their content is folded in Pass 1, and Pass 2 skips triples mentioning reifiers—the essential difference from DIRECT, where $(r_1, \mathit{statedBy}, \mathit{alice})$ was an edge of its own. The six nodes are the five IRIs and a Literal *node for 0.8: every description object is materialized so that the foreign keys in the folded records can reach it.*

Under open folding, each reifying triple contributes one triple term edge. The two reifiers of τ_1 thus yield two parallel madeOf edges from ID(sun) to ID$(\mathit{popcorn})$, distinguished only by their folded records—which retain the descriptions (alice, bob, 0.8) but no trace of r_1 or r_3 themselves. Both keep asserted $\mapsto$ `false`*, as τ_1 is never asserted; the edge for τ_2, created for the description-free r_2, is flipped to* `true` *by Pass 2 as in Example 2. Under fixed folding, the two descriptions of r_1 would instead occupy one edge each (via* refProperty/references*), giving four triple term edges rather than three.*

[4] Repeated predicates could be handled by list-valued properties. To keep our algorithms concise, we do not formalize or implement this detail.

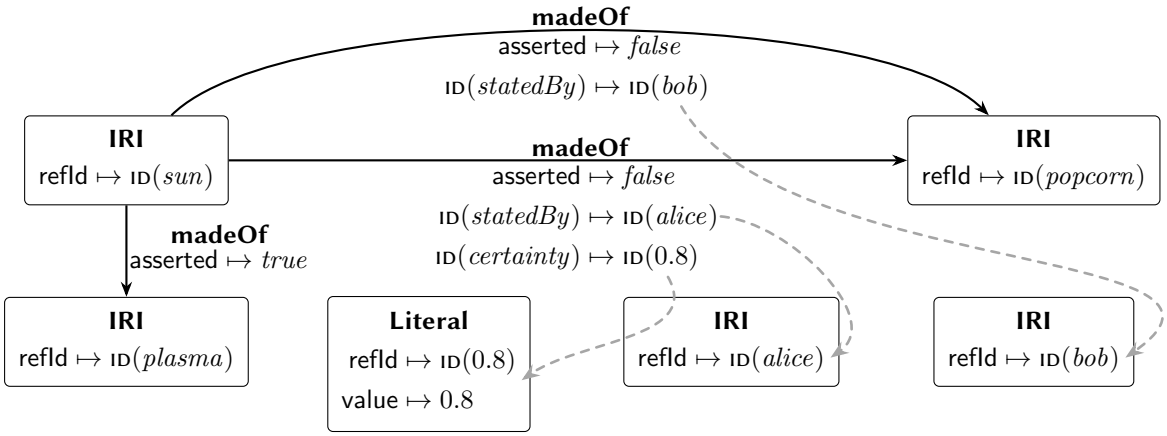


Figure 4: The Fold encoding of Example 3 with open folding.

Algorithm 3 also defines two passes. Pass 1 folds the descriptions of each reifier onto triple term edges for τ, all initially carrying asserted ↦ `false`: fixed folding creates one such edge per description, whereas open folding creates a single edge carrying every description. A reifier with no description creates one triple term edge carrying only asserted ↦ `false`. Pass 2 then visits every top-level triple whose subject and object are not reifiers. If it corresponds to a triple term edge, assertedness is flipped to asserted ↦ `true`; otherwise it becomes a regular (asserted) edge. A triple that mentions a reifier is skipped. Because a reifier is never a node, its descriptions were already folded in Pass 1.

3.4. Trade-offs

Table 1 summarizes the trade-off between completeness (i.e., which graphs a translation accepts, and what it preserves) and how closely its output realizes LPGs' native edge-with-properties construct. Call a translation *lossless* on a class of RDF graphs if the input can be reconstructed exactly from the output LPG; as node identifiers can be read back to RDF terms, this hinges only on the four rows of Table 1, that is, how triple terms, reifiers, descriptions, and assertedness are encoded. Structural is lossless on *every* RDF 1.2 graph: an edge without in is an asserted triple, and a TripleTerm node's triple is recovered, recursively, through its in edge. Direct is lossless on flat reification graphs, the class delimited by its two assumptions: each edge carrying reifiedBy decodes to its reifying triple, plus the plain triple iff asserted is `true`, and every other edge (descriptions included) decodes directly to its triple. Fold is lossy even on strict flat reification graphs, precisely at the reifiers: renaming r_1 in Figure 1 gives us the same LPG (the output in Example 3 carries no trace of r_1 or r_3), and a bare reifier of an asserted triple leaves no trace at all. Beyond these two effects nothing is lost under open folding; fixed folding additionally forgets which descriptions share a reifier, in exchange for its fixed record schema, where open folding's keys are data-dependent (Section 3.3). Size in Table 1 counts the elements (nodes and edges) of the output on inputs all variants accept.

4. Experimental Evaluation

We implemented all three translation algorithms of Section 3; for Fold, fixed and open folding are measured separately. The evaluation has two goals: (1) to demonstrate empirically that each variant converts RDF 1.2 data into labeled property graphs on its intended class of inputs, and (2) to test how the trade-offs predicted in Section 3.4 materialize in practice. We measure conversion time, the converter's memory usage (peak Resident Set Size (RSS)), and the size of the generated output.

For brevity, we omit methodological details (hardware and software configuration, repetitions per configuration, etc.); all reported numbers are averages over repeated runs. An extended version of

Algorithm 3 FOLD: reifies-interpreting translation with folded descriptions

Shared state: $N, E, \mathsf{edg}, \mathsf{lab}, \mathsf{rec}$; **parameter** $form \in \{fixed, open\}$

procedure FOLD(G)
  $N \leftarrow \emptyset$; $E \leftarrow \emptyset$; edg, lab, rec $\leftarrow$ empty maps
  reifiers $\leftarrow \{ r \mid (r, reifies, \tau) \in G,\ \tau \in \mathsf{Triples} \}$
  ttEdges $\leftarrow$ empty map ▷ maps triples to triple term edges
Pass 1: fold each reifier's descriptions onto triple term edges
  for all $(r, reifies, \tau) \in G$ **with** $\tau \in \mathsf{Triples}$ **do**
    descriptions $\leftarrow \{ (r, q, v) \in G \mid q \neq reifies \}$
    for all $(r, q, v) \in$ *descriptions* **do**
      ADDNODE(v) ▷ description object becomes a node
    for all *rec* $\in$ CREATERECORDS(*descriptions*) **do**
      $e \leftarrow$ CREATEEDGE(τ, *rec*)
      ttEdges(τ) $\leftarrow$ *ttEdges*(τ) $\cup \{e\}$
Pass 2: plain triples that mention no reifier
  for all $(s, p, o) \in G$ **with** $o \notin \mathsf{Triples}$ **and** $s, o \notin$ *reifiers* **do**
    if *ttEdges*$(s, p, o) \neq \emptyset$ **then**
      for all $e \in$ *ttEdges*(s, p, o) **do**
        rec(e)(asserted) $\leftarrow$ `true`
    else
      CREATEEDGE((s, p, o), {asserted $\mapsto$ `true`})
  return $(N, E, \mathsf{edg}, \mathsf{lab}, \mathsf{rec})$

procedure CREATERECORDS(*descriptions*) ▷ set of edge records for a reifier's descriptions, per *form*
  if $form = open$ **then** ▷ all descriptions on one record
    rec $\leftarrow$ {asserted $\mapsto$ `false`}
    for all $(r, q, v) \in$ *descriptions* **do**
      rec(ID(q)) $\leftarrow$ ID(v) ▷ predicate is the key
    return { *rec* }
  if *descriptions* $= \emptyset$ **then**
    return { {asserted $\mapsto$ `false`} } ▷ fixed, bare reifier
  return { {asserted $\mapsto$ `false`, refProperty $\mapsto$ ID(q), references $\mapsto$ ID(v)} $\mid (r, q, v) \in$ *descriptions* } ▷ fixed, one record per description

Table 1
Overview. *Flat graphs*: triple terms occur only as objects of *reifies* and are not nested; *strict*: additionally, reifiers occur only as subjects, each reifying a single triple term.

	STRUCTURAL	DIRECT	FOLD
Input graphs	all of RDF 1.2	flat graphs	strict flat graphs
Triple term	node + content edge	edge per reifying triple	edge(s) per reifying triple
Reifier	ordinary node	Reifier node	absent
Descriptions	ordinary edges	ordinary edges	folded into edge records
Assertedness	absence of in	asserted key	asserted key
Lossless	yes	yes, on its class	no: reifiers lost
Size (nodes + edges)	largest	intermediate	smallest

this section, the raw results of every measurement, and reproduction instructions are available in the accompanying repository[5].

Datasets Our main workload consists of samples of the RDF-star representation of the Biomedical Knowledge Repository (BKR) published with the REF benchmark [10] and used by StarBench [11] ; we

[5] https://github.com/RDF-PG-Interoperability/rdf12-to-pg/tree/cc328d1733502dabbe964f63551795ad9436b7d6/experiments

Table 2
Samples derived from the BKR/REF RDF-star data and migrated to RDF 1.2.

Target size	Reifiers	Descriptions	Descr./reifier	Turtle bytes	Graph triples
0.5 MB	154	8,835	57.4	499,932	8,989
1 MB	134	18,554	138.5	999,917	18,688
1.5 MB	159	28,042	176.4	1,499,694	28,201
2 MB	211	37,402	177.3	1,999,851	37,613
2.5 MB	286	46,639	163.1	2,499,756	46,925

evaluate conversion only, not query execution. In BKR, each embedded triple is a biomedical proposition annotated with *provenance* values: the sources from which the proposition derives. As the source uses the earlier RDF-star syntax, in which an embedded triple may occupy subject position, each record `<<s p o>> q v1, ..., vk` is rewritten with a fresh, deterministically named blank-node reifier r as `r rdf:reifies <<(s p o)>>` and `r q v1, ..., vk`. This retains the embedded triple's terms, its non-asserted status, and its provenance values, but changes the graph structure and query patterns.

We sampled five target sizes, from 0.5 MB to 2.5 MB in 0.5 MB increments (Table 2). The samples are deterministic but not nested prefixes of one another, so their composition varies with size. After migration, every provenance value becomes a description of its proposition's reifier, so the samples exhibit a high description count of 57 to 177 descriptions per reifier; we call such workloads *description-heavy*.

We additionally ran the implementations on a synthetic dataset covering many RDF 1.2 features, generated by a publicly available tool[6]. It includes standalone and nested triple terms, which fall outside the input restrictions of Direct and Fold (Section 3); their implementations skip such triples unless the predicate is `rdf:reifies`. Successful conversion of this workload therefore demonstrates operational robustness, not information preservation outside the stated domains. Details on these additional experiments can be found in the repository.

Output formats Each conversion outputs YARS-PG syntax [12] or a Neo4j loading script in Cypher, either *monolithic* (the entire graph in one transaction) or *batched* (1,000 rows per transaction, with a loader-specific indexed identifier that is not part of the conceptual translations). Every combination of translation, folding form, output format, and input size was executed.

Results All runs completed without errors. Figure 5 shows conversion time, peak RSS, and output size on the BKR workload. Four observations stand out.

First, all measurements grow approximately linearly, and monolithic conversion time is nearly identical across the algorithms (1.02–1.15 s at 2.5 MB): the generality of Structural has no discernible runtime penalty.

Second, the choice of output format matters more than the choice of translation. The batched scripts are designed for memory-constrained loading, and generating is a speed vs. memory trade-off: the converter tracks already-created nodes on disk rather than in memory, which for Direct and Structural slows generation (4.5 s and 3.5 s at 2.5 MB, versus roughly one second in monolithic mode) and enlarges the files, but more than halves their memory usage (≈50 MiB versus 124–134 MiB). The Fold implementation does less of this bookkeeping, so its batched generation stays fast (1.2–1.3 s), with correspondingly smaller memory savings.

Third, the folding strategy dominates output size on this description-heavy workload. Direct, Structural, and fixed folding all produce one edge per description, whereas open folding produces one triple term edge per reifier (at 2.5 MB, 286 edges in place of 46,639). Byte sizes (Table 3), however, do not simply follow these counts, since what each element records matters as much. Fixed folding,

[6] https://github.com/RDF-PG-Interoperability/rdf12-to-pg/tree/cc328d1733502dabbe964f63551795ad9436b7d6/synthetic-generator

Table 3
Output size at the 2.5 MB input (monolithic loading, in MB).

	Direct	Fold fixed	Fold open	Structural
Cypher	8.01	11.46	3.35	7.15
YARS-PG	7.55	11.00	3.17	6.63

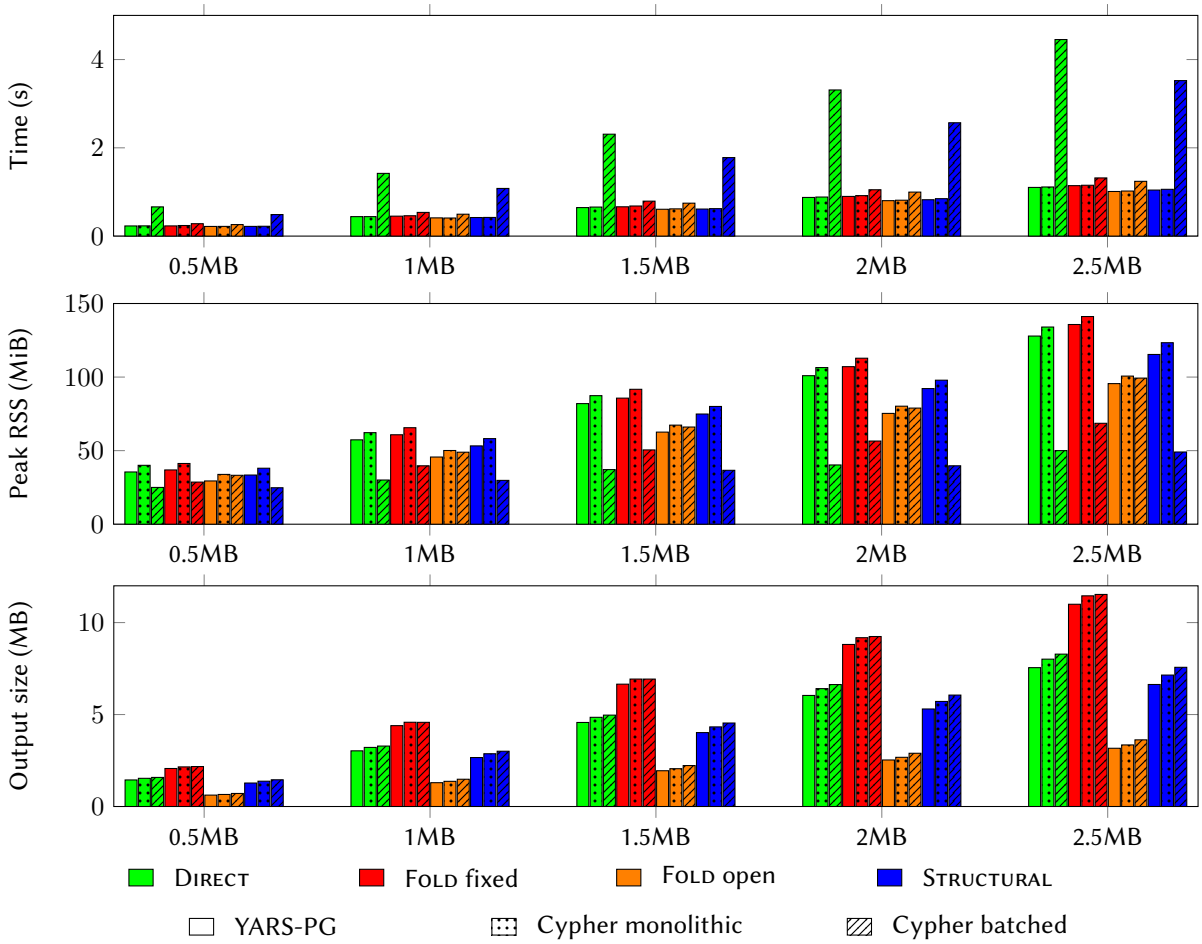


Figure 5: Conversion time, peak RSS, and output size for the BKR dataset; fixed and open folding are shown separately.

though smaller in nodes and edges than Direct and Structural, produces the largest files: every per-description edge repeats the reified triple's source, target, and label, which at 57–177 descriptions per reifier outweighs the reifier nodes and description edges it saves. The same effect ranks Structural below Direct: Direct marks every edge with the asserted flag, which on tens of thousands of edges outweighs the few hundred extra nodes Structural creates. Open folding results by far in the smallest output, though its measured bytes should be interpreted with caution: BKR's descriptions share one provenance predicate per reifier, and our prototype retains a single value per predicate key.[7]

Fourth, complementary measurements on the synthetic workload, reported in the repository, suggest that the batched scripts pay off downstream: in the one setting measured in both modes, loading the batched script into Neo4j took 6.97 s on average, compared with 345.16 s for the monolithic script, making it roughly 50 times faster.

In total, the experiments demonstrate the feasibility of all three translations and support the trade-offs discussed in Section 3.4.

[7]Instead, one could implement the list-valued extension of Section 3.3 to fully preserve information.

5. Conclusion

We studied the translation of RDF 1.2 graphs into LPGs, focusing on the statement-level constructs that LPGs lack. Our three translations differ in how much of the reification vocabulary they interpret: Structural interprets none of it and is lossless on all of RDF 1.2; Direct interprets `rdf:reifies` and is lossless on flat reification graphs while retaining reifiers as nodes; and Fold folds descriptions into edge records, the native LPG representation of annotated statements, at the cost of reifier identity. Our experiments show that all three are practical and corroborate the predicted trade-offs: the generality of Structural carries no discernible runtime penalty, making it the safest general-purpose choice; open folding makes Fold the variant of choice for description-heavy data, producing one annotated edge per reifier where the other configurations produces one per description;

Several directions remain open. A natural next step is to complement structural losslessness with *query losslessness*: translating SPARQL 1.2 queries into GQL queries over the translated LPG and validating answer equivalence, thereby characterizing each translation by the queries it preserves rather than the graphs it reconstructs. On the representation side, literal-valued descriptions could be inlined as plain record values rather than materialized as referenceable nodes, improving both conciseness and performance. Finally, we plan to lift the translations from RDF graphs to RDF datasets with named-graph support, and to revisit aspects left out of scope here: entailment, native blank-node handling instead of Skolemization, and vendor-specific features of concrete graph database systems.

Acknowledgments

This publication is based upon work performed during a hackathon of COST Action CA23147 GOBLIN — Global Network on Large-Scale, Cross-domain and Multilingual Open Knowledge Graphs, supported by COST (European Cooperation in Science and Technology).

Declaration on Generative AI

During the preparation of this work, the authors used Claude Fable 5 for: Drafting content, Peer review simulation, Content enhancement, and Formatting assistance. The authors extensively reviewed and edited the content as needed and assume full responsibility for the content of the publication.

References

[1] O. Hartig, Foundations of rdf⋆ and sparql⋆ (an alternative approach to statement-level metadata in RDF), in: AMW, volume 1912 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2017.

[2] G. Kellogg, O. Hartig, P.-A. Champin, A. Seaborne, RDF 1.2 Concepts and Abstract Data Model, Technical Report, W3C, 2026. URL: https://www.w3.org/TR/rdf12-concepts/.

[3] A. Gschwend, O. Lassila, RDF & SPARQL Working Group Charter, Technical Report, W3C, 2025. URL: https://w3c.github.io/rdf-star-wg-charter/.

[4] R. Angles, H. Thakkar, D. Tomaszuk, Mapping RDF databases to property graph databases, IEEE Access 8 (2020) 86091–86110.

[5] K. Rabbani, M. Lissandrini, A. Bonifati, K. Hose, Transforming RDF graphs to property graphs using standardized schemas, Proc. ACM Manag. Data 2 (2024) 242:1–242:25.

[6] O. Hartig, Reconciliation of rdf* and property graphs, CoRR abs/1409.3288 (2014).

[7] S. Das, J. Srinivasan, M. Perry, E. I. Chong, J. Banerjee, A tale of two graphs: Property graphs as RDF in oracle, in: EDBT, OpenProceedings.org, 2014, pp. 762–773.

[8] G. Abuoda, D. Dell'Aglio, A. Keen, K. Hose, Transforming RDF-star to Property Graphs: A Preliminary Analysis of Transformation Approaches, in: QuWeDa@ISWC, volume 3279 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2022, pp. 17–32.

[9] ISO/IEC, Information Technology—Database Languages—GQL, International Standard ISO/IEC 39075:2024, International Organization for Standardization, Geneva, Switzerland, 2024. URL: https://www.iso.org/standard/76120.html.

[10] F. Orlandi, D. Graux, D. O'Sullivan, Rdf reification benchmark (ref) using the biomedical knowledge repository (bkr), 2020. URL: https://doi.org/10.5281/zenodo.4148888.

[11] G. Abouda, C. Aebeloe, D. Dell'Aglio, A. Keen, K. Hose, Starbench: Benchmarking rdf-star triplestores, in: # PLACEHOLDER_PARENT_METADATA_VALUE#, volume 3565, CEUR-WS. org, 2023, pp. 34–49.

[12] D. Tomaszuk, R. Angles, L. Szeremeta, K. Litman, D. Cisterna, Serialization for property graphs, in: BDAS, volume 1018 of *Communications in Computer and Information Science*, Springer, 2019, pp. 57–69.