Using Provenance to Efficiently Propagate SPARQL Updates on RDF Source Graphs

Iman Naja(✉) and Nicholas Gibbins

Electronics and Computer Science, University of Southampton, Southampton, UK
{i.naja,nmg}@ecs.soton.ac.uk

Abstract. To promote sharing on the Semantic Web, information is published in machine-readable structured graphs expressed in RDF or OWL. This allows information consumers to create graphs using other source graphs. Information, however, is dynamic and when a source graph changes, graphs based on it need to be updated as well to preserve their integrity. To avoid regenerating a graph after one of its source graphs changes, since that approach can be expensive, we rely on its provenance to reduce the resources needed to reflect changes to its source graph. Accordingly, we expand the W3C PROV standard and present RGPROV, a vocabulary for RDF graph creation and update. RGPROV allows us to understand the dependencies a graph has on its source graphs and facilitates the propagation of the SPARQL updates applied to those source graphs through it. Additionally, we present a model that implements a modified DRed algorithm which makes use of RGPROV to enable partial modifications to be made on the RDF graph, thus reflecting the SPARQL updates on the source graph efficiently, without having to keep track of the provenance of each triple. Hence, only SPARQL updates are communicated, the need for complete re-derivation is done away with, and provenance is kept at the graph level making it better scalable.

Keywords: Provenance · PROV · RDF · SPARQL update

1 Introduction

The Semantic Web promotes the publishing, understanding, discovery, integration, and re-use of information, with recent years seeing a boost in the publication, inter-linkage, and consumption of large amounts of public datasets. Knowledge is presented in machine-understandable formats, namely RDF [1] and OWL [2] graphs, which provide well-defined meanings and support rules for reasoning, and is queried and updated using SPARQL [3]. Graphs may be manually created or automatically formed by combining information from other graphs, and automated reasoning may be performed on them.

However, this is not without challenge, as knowledge is neither static nor complete and its expansion and change is inevitable. Thus, in systems having

K. Belhajjame et al. (Eds.): IPAW 2018, LNCS 11017, pp. 158–170, 2018.
https://doi.org/10.1007/978-3-319-98379-0_12

graphs which relied on other source graphs when created, changes to those source graphs need to be incorporated and reflected so as to keep such graphs up-to-date. Typically, systems recreate those graphs from scratch and reason anew on them. This may be expensive, and sometimes impractical to re-obtain the data used and to re-reason with it. Alternatively, a system may contain its own reasoner which takes responsibility for re-reasoning, like in [4].

Another challenge arises in the fact that the Semantic Web is an open environment where 'anyone can say anything about anything'. This begets the need for means to appraise the trustworthiness, reliability, and reputation of data in graphs to be consumed; and such assessments are intrinsically linked to knowing their provenance. Provenance describes the history of a datum or thing, and which activities, entities, and people were involved in how they came to be [5]. It has proven to be useful in numerous domains, as developers, researchers, and users have been utilising it to establish trust, understanding, transparency, attribution and accountability for outputs of intelligent systems. Moreover, the recent community-driven work to achieve an open provenance vision resulted in the PROV data model [6], a W3C recommendation.

While PROV facilitates interoperable provenance modelling, it is generic; a more specialised vocabulary better serves to track and express the provenance specific to RDF graphs, relating their creation and detailing and facilitating their modification. Accordingly, we expand PROV and present RGPROV, a vocabulary which models the classes and properties involved in an RDF graph's creation and update. It allows the specific capture of the provenance of an RDF graph created using other graphs and understanding its dependencies on them. It also expedites the propagation of SPARQL updates applied to its source graphs without wide scale insertions or deletions and then complete re-derivation, thus promoting the capture of the provenance of the update precisely and efficiently, without resorting to tracking the provenance of individual triples.

The contributions of this paper are fourfold. (*i*) Our main contribution is the RGPROV vocabulary, a specialisation of PROV-O which models the classes and properties involved in an RDF graph's creation and the SPARQL updates applied on it. (*ii*) A partial re-derivation algorithm, based on DRed [7], which makes use of RGPROV to propagate all or some of the SPARQL updates applied on source graphs. (*iii*) A model which implements both RGPROV and the partial re-derivation algorithm and (*iv*) A quantitative evaluation of our model demonstrating that less resources are needed to achieve the same results.

Outline: Sect. 2 presents related work. Section 3 provides the running example used throughout. Section 4 presents RGPROV. Section 5 describes our model and presents the partial re-derivation algorithms. Section 6 describes the implementation of our system and presents the results. Finally, Sect. 7 presents our conclusions and future work.

2 Related Work

The Delete and Rederive (DRed) algorithm [7] deletes the base data and all the data that was derived from it, then re-inserts the subset of the derived data

that can be re-derived using other still present base data. RDFox [4] initially materialises queries and its reasoner implements an incremental maintenance algorithm based on DRed, but without making use of provenance. Elseways, [9] presents an initial model, where they aim to have versioned data and functions which would use provenance to detect changes between data versions, select processes that require re-computation, and decide between complete or partial re-computation. [10] extends their work on provenance semirings in [11] to support update exchange, schema mapping, and trust evaluation and to also extend the DRed algorithm. When a deletion occurs, they utilise provenance to flag and delete tuples which are no longer derivable. Similarly, [12] extends [13]'s work and utilises colours to represent triple sources, although they consider inferred quadruples independent of their sources. Before a quadruple is deleted, all the quadruples that can be inferred from it are inserted first. Then, all the quadruples that would entail it are deleted along with the quadruple itself. Further, [14] extends both [11,13]'s work by also using quadruples. Their quadruples' fourth elements are named graphs and quadruples' provenance is maintained in separate tuples with an id element linking to them. They, however, do not consider deletions; their algorithm describes how to insert quadruples and record their provenance. The aforementioned works track provenance on the triple level, an approach we avoided because of scalability concerns since tracking each triple's provenance using PROV would result in a graph having the size of its provenance graph substantially larger than it[1]. Instead, using RGPROV we track provenance on the graph level. [15] presents work similar to ours that tracks dynamic provenance of collections using a specialisation of PROV, upd, which allows them to capture SPARQL queries and updates performed on raw data in a dataset. Their work only considers updates and ignores the other operations that may affect a graph or its provenance, namely fetching, set theoretic operations, and re-entailment.

3 Running Example

We assume there are four systems A, B, C, and D, as shown in Fig. 1, each having ownership of some RDF graphs and maintaining their provenance. We focus on system C and explain our notations whilst identifying activities performed on graphs. A graph $G_{X,n}$ belongs to system X, differentiated from other graphs by the subset n, and $P_{X,n}$ is its provenance graph.

(*1*) *Graph retrieval:* the activity of fetching a graph and its provenance from an external system and saving their copies internally. When copied to system C, $G_{X,n}$'s name becomes $G_{copy(X,n)}$ in C. Similarly $P_{X,n}$'s name becomes $P_{copy(X,n)}$. To reflect the activity of copying, $P_{copy(X,n)}$ is updated and becomes $P^*_{copy(X,n)}$.

[1] If each triple's provenance consists of only `triple prov:wasDerivedFrom sourceTriple`, a graph's provenance graph would be a little larger than it. Even adding provenance information about only the activity and agent that produced a triple would result in the graph's provenance graph being at minimum triple its size.

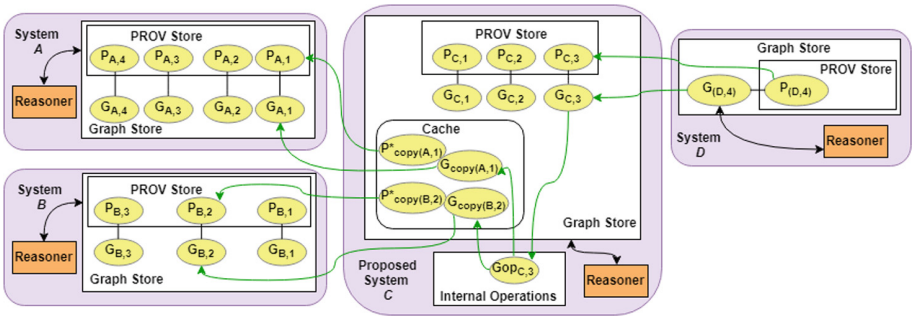

Fig. 1. Example of distributed graph usage.

(*2*) *Set theoretic operations:* An intermediary graph $Gop_{C,m}$ is produced by applying one of Union, Merging, Intersection, and Difference. We currently ignore blank nodes, and subsequently Merging.
(*3*) *Entailment:* In C, the entailed graph $G_{C,m}$ is produced from $Gop_{C,m}$ by running it through a reasoner. Entailment operations depend on which entailment regime is implemented, namely: RDF, RDFS, Datatype, OWL 2 RDF-Based Semantics, OWL 2 Direct Semantics, or RIF. We use RDFS Entailment [8] in our system.
An example of $G_{C,3}$'s production is shown in Fig. 2. Throughout the identified activities, C produces and updates the provenance $P_{C,3}$ of $G_{C,3}$. Note that the aforementioned list of operations to create a graph is not exhaustive; other operations, including the use of join, CONSTRUCT, OPTIONAL, etc., are beyond the scope of this paper and may be addressed in some future work.
(*4*) *SPARQL updates:* If a system, say B, performs a SPARQL update $Up_{op(B,2)}$ on $G_{B,2}$ resulting in it becoming the new graph $G_{B',2}$, then C should know about this update and subsequently needs to update $G_{C,3}$, or whichever parts of it should be affected, thus resulting in the more accurate and up-to-date $G_{C',3}$. SPARQL updates are Insert, Delete, Delete/Insert, Load, and Clear. We only focus on Insert and Delete as the latter three can be seen as combinations or special cases of the former. The standard approach is to retrieve a copy of $G_{B',2}$ and $G_{A,1}$ - if not internally stored, reapply *Gop* on them, and re-entailing to produce $G_{C',3}$. This becomes impractical in large systems for two reasons: (1) it is computationally expensive to re-entail a sizeable graph from scratch whenever there is an update, and (2) it requires additional storage, communication overhead, or both since the source graphs either need to be stored or re-fetched whenever a change occurs. Thus, we identify the need for a more efficient way to reflect updates and produce $G_{C',3}$. Our approach considers both the set theoretic operation which created $Gop_{C,3}$ and the nature of $Up_{op(B,2)}$. This allows us to retrieve only the update $Up_{B,2}$ applied to $G_{B,2}$ instead of retrieving all of $G_{B',2}$ and to identify whether $G_{copy(A,1)}$ is required, whether all or part of $Up_{B,2}$ needs to be propagated, and which parts of $G_{C,3}$ need to be re-derived.

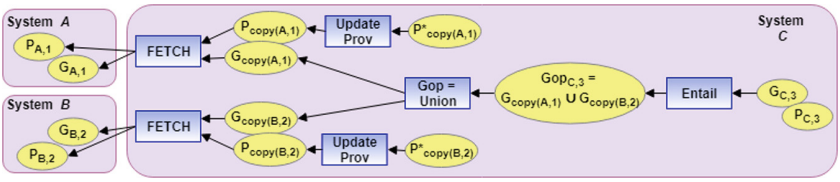

Fig. 2. Production of $G_{C,3}$ from $G_{A,1}$ and $G_{B,2}$.

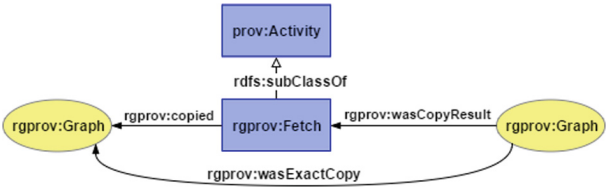

Fig. 3. RGPROV components for graph retrieval.

4 The RGPROV Vocabulary

RGPROV extends PROV-O and has the namespace prefix rgprov. Although we only use it for RDF graphs, we intend it to be used for OWL graphs as well.

In accordance with PROV, we recognize that RDF and PROV graphs are entities. To differentiate them from other types of entities, we introduce the class *Graph*, a subclass of prov:Entity, that contains only entities which are graphs. The actions that retrieve, produce, or update a Graph are activities, initiated by agents. We extend these concepts and any necessary properties as follows.

Vocabulary for Graph Retrieval: We require stricter terms than prov:hadPrimarySource and prov:wasQuotedFrom to represent copying a graph as-is from its sources. Based on the description in Sect. 3, we show them in Fig. 3. We see no need to create additional vocabulary for provenance production and updating because provenance graphs are members of Graph, hence RGPROV's terms can be adequately applied to them.

Vocabulary for Graph Operations: We introduce the class *GraphOperation*, a subclass of prov:Activity, that encompasses operations performed on a graph.

Vocabulary for Set theoretic Operations: Because there is a need to keep track of which graph operation produced a graph, we introduce terms for set theoretic operations, based on the description in Sect. 3, and show them in Fig. 4.

Vocabulary for Entailment Operations: To describe entailment operations, we introduce the following, based on the description in Sect. 3, and depict a selection of them in Fig. 5:

(1) *Entailment*, a subclass of GraphOperation, with subclasses representing particular entailment regimes.

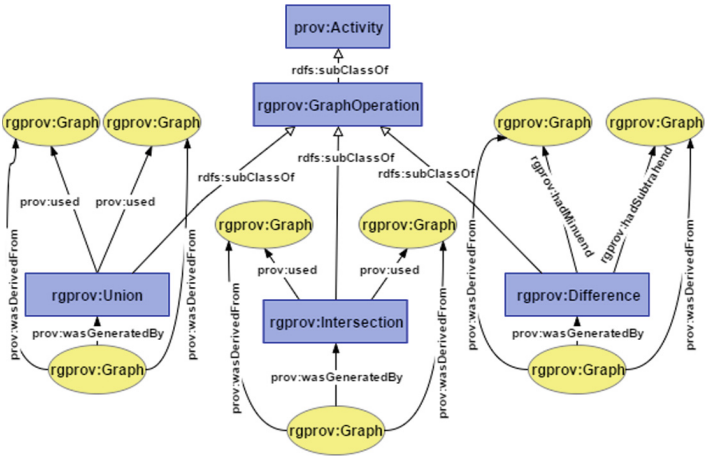

Fig. 4. RGPROV components for set theoretic graph operations.

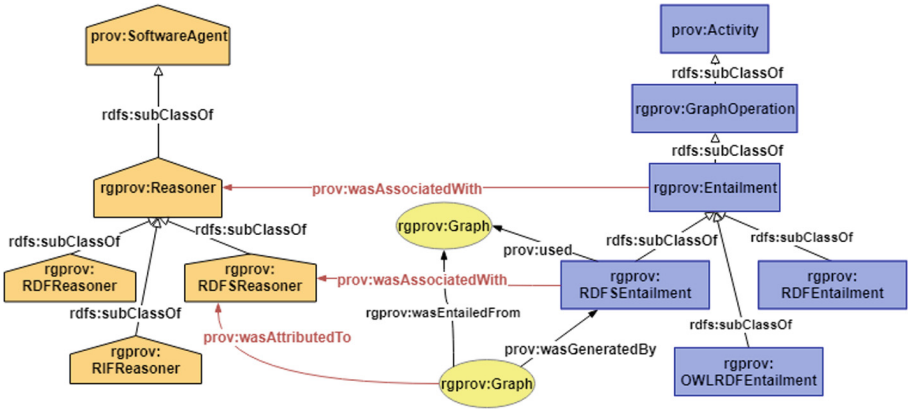

Fig. 5. Some RGPROV components for entailment operations.

(2) *Reasoner*, a subclass of prov:SoftwareAgent that represent a reasoner, with subclasses representing reasoners performing particular entailment regimes.
(3) *wasEntailedFrom*, a subproperty of prov:wasDerivedFrom, has domain Graph, has range Graph.

Vocabulary for Updates: First, we introduce *UpdateGraph*, a subclass of Graph that represents the graphs whose triples are to be inserted or deleted. We argue for this because a graph that is stored in and being used by a system should be differentiated from one whose entire purpose is containing triples to be inserted or deleted in the former type of graph. Additionally, since we differentiate the types of updates performed on a graph, we require stricter terms than prov:Revision

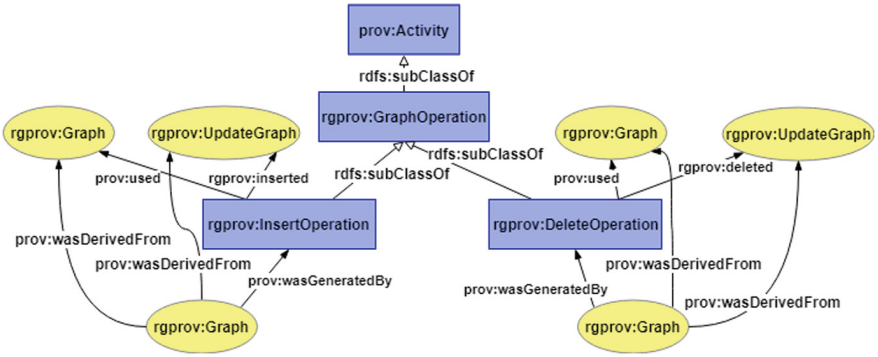

Fig. 6. RGPROV components for update operations.

and prov:wasRevisionOf. Thus we introduces terms for graph updates based on the description in Sect. 3, and show them in Fig. 6.

RGPROV is published on https://archive.org/download/rgprov/rgprov.owl and https://archive.org/download/rgprov/rgprovTurtle.owl.

5 The Model and Algorithms

5.1 System Architecture

We designed a system, shown in Fig. 7, comprising seven components, of which we implemented four. (*1*) Operator, the main component, responsible for controlling and invoking the operations performed on the graphs in the system. As the central component, it invokes and communicates with the other components. (*2*) Provenance Handler, responsible for creating, querying, and updating provenance graphs. (*3*) SPARQL Server and Graph Store, which we have not implemented but used the third party Jena Fuseki Server[2]. (*4*) Reasoner, which we have also not implemented but used the third party Jena[3]. Jena is responsible for performing the set theoretic and entailment operations on all graphs. (*5*) Update Producer, handles any updates applied on graph $G_{C,3}$ for any outside system that uses it. (*6*) Cache, used to store copies of retrieved graphs or updates and any other temporary graphs as needed. Finally, (*7*) REST client, handles the communications between the different systems. We have not implemented it, as it does not pertain to the demonstrating the application of the RGPROV vocabulary nor does it affect the evaluation of the system. Components have been implemented in Java.

Note that unless they have been marked as inferred triples, all triples in the source graphs are treated as ground triples in the system. Then, after it is

[2] Fuseki2 is available on https://jena.apache.org/documentation/fuseki2/.

[3] All Jena binary distributions are available on http://archive.apache.org/dist/jena/binaries/.

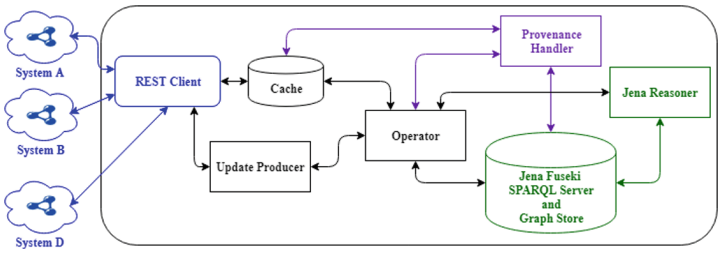

Fig. 7. System design.

produced, graph $G_{C,3}$ is split into and stored as two graphs. The first consists of the ground triples and the second consists of the inferred triples produced by our systems' reasoner. This separation proves beneficial when re-deriving to minimise over-deletions and re-insertions.

5.2 Update Propagation per Set Theoretic Operations

We now analyse how the combination of the set theoretic operation and the kind of update influence what part of the triples in the update graph $Up_{B,2}$ inserted into or deleted from $G_{B,2}$ are to be propagated into graph $G_{C,3}$ and how. Note that inserting triples which already exist in a graph has no effect, nor does deleting triples which do not exist.

Union $G_{C,3} = G_{copy(A,1)} \cup G_{copy(B,2)}$.
Insert: equivalent to inserting into $G_{C,3}$ the triples in $Up_{B,2}$. $G_{copy(A,1)}$ is not needed and the only new entity needed is $Up_{B,2}$.
Delete: equivalent to deleting from $G_{C,3}$ the triples in $Up_{B,2} \setminus G_{copy(A,1)}$. $G_{copy(A,1)}$ is needed along with $Up_{B,2}$.
Intersection $G_{C,3} = G_{copy(A,1)} \cap G_{copy(B,2)}$.
Insert: equivalent to inserting into $G_{C,3}$ the triples in $Up_{B,2} \cap G_{copy(A,1)}$. $G_{copy(A,1)}$ is needed along with $Up_{B,2}$.
Delete: equivalent to deleting from $G_{C,3}$ the triples in $Up_{B,2}$. $G_{copy(A,1)}$ is not needed and the only new entity needed is $Up_{B,2}$.
Difference Case 1 $G_{C,3} = G_{copy(A,1)} \setminus G_{copy(B,2)}$.
Insert: equivalent to deleting from $G_{C,3}$ the triples in $Up_{B,2}$. $G_{copy(A,1)}$ is not needed and the only new entity needed is $Up_{B,2}$.
Delete: equivalent to inserting into $G_{C,3}$ the triples in $Up_{B,2} \cap G_{copy(A,1)}$. $G_{copy(A,1)}$ is needed along with $Up_{B,2}$.
Difference Case 2 $G_{C,3} = G_{copy(B,2)} \setminus G_{copy(A,1)}$.
Insert: equivalent to inserting into $G_{C,3}$ the triples in $Up_{B,2} \setminus G_{copy(A,1)}$. $G_{copy(A,1)}$ is needed along with $Up_{B,2}$.
Delete: equivalent to deleting from $G_{C,3}$ the triples in $Up_{B,2}$. $G_{copy(A,1)}$ is not needed and the only new entity needed is $Up_{B,2}$.

5.3 Partial Re-derivation Algorithms

The Operator queries $P_{C,3}$ for the set theoretic operation that produced $G_{C,3}$, checks the update type, and cross-references that pair with the list in Sect. 5.2 to decide whether all the update graph $Up_{copy(B,2)}$ or a subset of it, namely $Subs_{Up_{copy(B,2)}}$, is to be applied to $G_{C,3}$. Before propagating the update, it removes the triples already present in the inferred portion of $G_{C,3}$, so as to avoid over-insertions/over-deletions and re-insertions. Finally, it applies the update as follows.

If the update is an insert, the Operator creates an Insert statement and sends it to Fuseki to be loaded into $G_{C,3}$. It then requests the graphs resulting from the Describe of those triples. The SPARQL Describe of a triple 'describes' it by returning a graph containing all those triples that are connected to it, i.e., all the triples which have as a subject any of the IRIs of the described triple's subject, predicate, or object. The union of the triples to be inserted and their descriptions constitute the entirety of information that is needed for re-derivation. This union is then forwarded to the reasoner, Jena, for inference. When the entailed graph resulting from reasoning on this union is returned, the Operator creates another Insert statement containing those inferred triples to be added by Fuseki thus resulting in $G_{C',3}$. The aforementioned is shown in Algorithm 1.

Algorithm 1. Apply Insert Update

Function: *describe* : *graph* x *triple* → *graph*
Function: *entail* : *graph* → *graph*

procedure APPLYINSERTUPDATE(*graph*, *triplesTBI*)
  described ← ϕ
  graph ← *graph* ∪ *triplesTBI*
  for each *triple* in *triplesTBI* **do**
    described ← *described* ∪ *describe*(*graph*, *triple*)
  graph ← *graph* ∪ *entail*(*described*)

If the update is a delete, then the Operator first gets, from Fuseki, the graphs resulting from the Describe of the triples to be deleted. It then loops over each triple to be deleted and examines its predicate. If the predicate is an rdf:type or has super-properties (i.e. it is a sub-property of another property), then it adds, to the list of the triples to be deleted, the triples with the same subject and any objects that relate it to the predicate. This is in accordance with the RDFS entailment rules described in [8]. Next, the Operator sends a Delete statement containing the updated list of triples to Fuseki, so that the latter deletes them from $G_{C,3}$, thus resulting in $Gop_{C',3}$. Afterwards, the Operator requests the Describe of all the subjects and objects that were in the deleted triples and sends the union of the resulting graphs to Jena for reasoning. When the entailed graph resulting from reasoning on the union is returned, the Operator sends

Algorithm 2. Apply Delete Update

Function: *describe* : *graph* x *triple* → *graph*
Function: *entail* : *graph* → *graph*

procedure ApplyDeleteUpdate(*graph*, *triplesTBD*)
  $described \leftarrow \phi$
  for each *triple* in *triplesTBD* **do**
    $described \leftarrow described \cup describe(graph, tripleTBD)$
  for each *triple* in *triplesTBD* **do**
    $subject = triple.Subj$
    $t \leftarrow \{t \in described | t.Subj = triple.Subj \wedge t.Prop = t.Prop\}$
    for each $t2$ in t **do**
      if $t2.Prop = rdf : type$ **then**
        $superClasses \leftarrow \{tsOAsS.Obj | tsOAsS \in described$
            $\wedge tsOAsS.Subj = t2.Obj \wedge tsOAsS.Prop = rdfs : subClassOf\}$
        for each *superClass* in *superClasses* **do**
          $infTriple \leftarrow \langle t2.Subj, rdf : type, superClass\rangle$
          $triplesTBD \leftarrow triplesTBD \cup infTriple$
      else
        $superProps \leftarrow \{tp.Obj | tp \in described \wedge tp.Subj = t2.Prop$
            $\wedge tp.Prop = rdfs : subPropertyOf\}$
        for each *superProp* in *superProps* **do**
          $infTriple \leftarrow \langle t2.Subj, superProp, t2.Obj\rangle$
          $triplesTBD \leftarrow triplesTBD \cup infTriple$
  $graph \leftarrow graph \setminus triplesTBD$
  ▷ Re-derive and insert inferred triples.
  $subjsAndObjs \leftarrow \{iri | iri \in triplesTBD.Subjects \cup triplesTBD.Objects\}$
  for each *iri* in *subjsAndObjs* **do**
    $described2 \leftarrow described2 \cup describe(graph, iri)$
  $graph \leftarrow graph \cup entail(described2)$

an Insert statement containing the inferred triples to Fuseki which inserts them thus producing $G_{C',3}$. The aforementioned is shown in Algorithm 2[4].

6 Results

To test our system, we created a small RDF-Schema to represent fictional characters and places, with both $G_{A,1}$ and $G_{B,2}$ making use of it. There are 12 classes and 35 properties. In addition to this schema, both graphs $G_{A,1}$ and $G_{B,2}$ contain instances of fictional characters and places. Graph $G_{A,1}$ contains 275 triples, while graph $G_{B,2}$ contains 265 triples. 15 triples are inserted into $G_{B,2}$ and then 4 triples are deleted from it. Table 1 displays the sizes of the produced graphs.

The evaluation criteria intends to verify that there is less overhead, in terms of the number of triples being processed, when performing the following:

[4] Due to space restrictions, the preceding description and subsequent algorithm only focus on ***rdfs 5***, ***7***, ***9***, and ***11***. Expanding them to cover the rest is straightforward.

Table 1. Size of $G_{C,3}$ initially, after Insert, and after Delete.

ST\sizes	Initial	Entailed	After Insert	Entailed	After Delete	Entailed
Union	355	865	369	913	366	905
Intersection	185	336	186	341	185	338
Difference 1	90	109	89	108	90	108
Difference 2	71	75	94	98	91	95

1. **Communication:** retrieving the update is less overhead than retrieving both source graphs.
2. **Execution:** propagating the update results in less triples processed during: (a) the set theoretic operation, and (b) re-derivation.

Experimental Results. There is indeed less overhead in applying our approach as detailed below.

(a) Communication. When the update is an Insert, the size of update will always be less than the size of the whole graph. Hence, there is less communication overhead. However, when the update is a Delete, the overhead of communicating the update is acceptable unless more than half of the triples in the graph are to be deleted, because the size of the update is greater than the size of the new source graph, and it may be more preferable to retrieve $G_{B',2}$ rather than $Up_{B,2}$. As shown in the analysis of the update propagation in Sect. 5.2, in the cases of intersection and the second difference, there is no need for $G_{copy(A,1)}$, but retrieving $G_{B',2}$ will force the re-retrieval of $G_{A,1}$ - if it is not stored in the system - plus the generation of $G_{C',3}$ from scratch. Combined, this would cause more overhead depending on the availability and the comparative size of $G_{A,1}$. Hence, retrieving $G_{B',2}$ would be more preferable. In the cases of union and the first difference where $G_{copy(A,1)}$ is needed, it may be more beneficial to retrieve $G_{B',2}$ instead of $Up_{B,2}$. So, it boils down to a case-by-case bases and can be alleviated by requesting the size of the update from system B and depends on if the other source graph is needed as well.

(b) Execution:

i. Set theoretic operations: We were not able to use Jena to count the triples processed in set theoretic operations. However, from our analysis in Sect. 5.2, we see that there are less triples to be checked because we are at most using the whole update and one source graph and not the entirety of both source graphs.

ii. Re-derivation: Inserting or deleting part of the update and then re-deriving by only taking into account the affected triples and those related to them reduces the number of triples processed by the reasoner in our experimental example by: 53% and 77% for the Union, 78% and 83% for the Intersection, 67% and 68% for the Difference 1, and 48% and 64% for the Difference 2. We point out that these gains may fluctuate depending on the triples chosen for insertion and deletion.

7 Conclusion and Future Work

We examined where provenance of graphs on the Semantic Web should be tracked, from their initial creation and through their modification, and based on this we introduced a specialisation of the PROV ontology, RGPROV. Then, we looked into how an update on a source graph needs to be propagated in a graph which is based on it and applied RGPROV to do so efficiently. Finally, we showed that our approach reduces overhead, in terms of number of processed triples, when compared to re-creating a graph from scratch by implementing a system which utilises algorithms to partially re-derive graphs.

There are a few directions worth exploring that extend our work. First, we aim to test our approach using benchmark data like LUBM[5] and UOBM[6]. Second, our system and re-derivation algorithms can be extended to support OWL graphs as well as RDF graphs by including OWL 2 entailment rules in the deletion and re-entailment phases. Finally, they may be extended to deal with source graphs which use different entailment regimes.

References

1. Schreiber, G., Raimond, Y.: RDF 1.1 primer. W3C note, W3C, June 2014. http://www.w3.org/TR/2014/NOTE-rdf11-primer-20140225/
2. Krötzsch, M., Patel-Schneider, P., Hitzler, P., Parsia, B., Rudolph, S.: OWL 2 web ontology language primer (second edition). Technical report, W3C, December 2012. http://www.w3.org/TR/2012/REC-owl2-primer-20121211/
3. SPARQL 1.1 overview. W3C recommendation, W3C, March 2013. http://www.w3.org/TR/2013/REC-sparql11-overview-20130321/
4. Motik, B., Nenov, Y., Piro, R., Horrocks, I.: Incremental update of datalog materialisation: the backward/forward algorithm. In: Proceedings of the 29th AAAI Conference on Artificial Intelligence, pp. 1560–1568. AAAI Press (2015)
5. Moreau, L., Groth, P.: PROV-overview. W3C note, W3C, April 2013. http://www.w3.org/TR/2013/NOTE-prov-overview-20130430/
6. Lebo, T., Sahoo, S., McGuinness, D.: PROV-o: the PROV ontology. W3C recommendation, April 2013. http://www.w3.org/TR/2013/REC-prov-o-20130430/
7. Gupta, A., Mumick, I.S., Subrahmanian, V.S.: Maintaining views incrementally. ACM SIGMOD Rec. **22**(2), 157–166 (1993)
8. Hayes, P., Patel-Schneider, P.: RDF 1.1 semantics. W3C recommendation, W3C, February 2014. http://www.w3.org/TR/2014/REC-rdf11-mt-20140225/
9. Missier, P., Cala, J., Wijaya, E.: The data, they are a-changin'. In: 8th USENIX Workshop on the Theory and Practice of Provenance (TaPP 2016). USENIX Association, Washington, D.C. (2016)
10. Green, T.J., Karvounarakis, G., Ives, Z.G., Tannen, V.: Update exchange with mappings and provenance. In: Proceedings of the 33rd International Conference on Very Large Data Bases, Vienna, Austria, pp. 675–686 (2007)
11. Green, T.J., Karvounarakis, G., Tannen, V.: Provenance semirings. In: Proceedings of the 26th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, Beijing, China, pp. 31–40. ACM (2007)

[5] http://swat.cse.lehigh.edu/projects/lubm/.
[6] https://www.cs.ox.ac.uk/isg/tools/UOBMGenerator/.

12. Flouris, G., Fundulaki, I., Pediaditis, P., Theoharis, Y., Christophides, V.: Coloring RDF triples to capture provenance. In: Bernstein, A., et al. (eds.) ISWC 2009. LNCS, vol. 5823, pp. 196–212. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-04930-9_13
13. Buneman, P., Cheney, J., Vansummeren, S.: On the expressiveness of implicit provenance in query and update languages. ACM Trans. Database Syst. **33**(4), 1–47 (2008)
14. Avgoustaki, A., Flouris, G., Fundulaki, I., Plexousakis, D.: Provenance management for evolving RDF datasets. In: Sack, H., Blomqvist, E., d'Aquin, M., Ghidini, C., Ponzetto, S.P., Lange, C. (eds.) ESWC 2016. LNCS, vol. 9678, pp. 575–592. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-34129-3_35
15. Halpin, H., Cheney, J.: Dynamic provenance for SPARQL updates. In: Mika, P., et al. (eds.) ISWC 2014. LNCS, vol. 8796, pp. 425–440. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-11964-9_27