SPARQL Update Processing: Extracting Inserted and Deleted Quads

Cristiano Aguzzi, Luca Roffia
University of Bologna
Bologna, Italy
{cristiano.aguzzi,luca.roffia}@unibo.it

***Abstract*—This short paper presents a novel algorithm to extract the inserted and deleted quads from a SPARQL 1.1 Update operation. The aim is to enable smarter approaches in the subscriptions processing of the SPARQL Event Processing Architecture (SEPA). We expect that the proposed algorithm would increase the overall SEPA performance by filtering out not affected subscriptions and optimizing the processing of each single subscription.**

I. Introduction

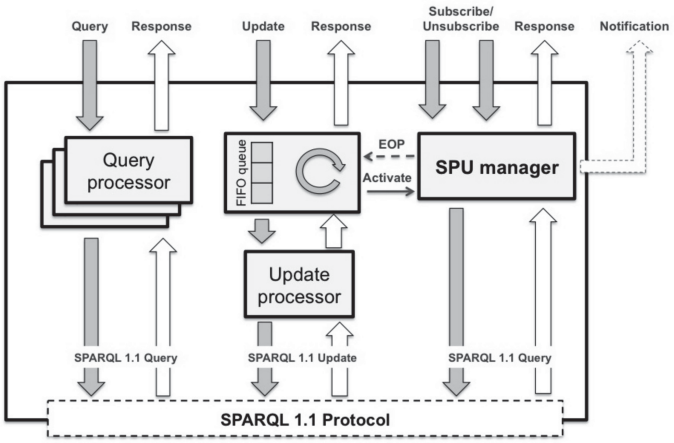
In the field of Linked Data [1], dealing with the dynamicity of data is a crucial aspect. At best of our knowledge the first attempts to tackle and formalize this problem were published in [2], while earlier results can be also found in [3] and [4]. In the past years, a number of technologies have emerged to manage the changes and retrieve meaningful information in this huge set of linked data. For example, C-SPARQL [5], SPARQLStream [6], event processing SPARQL (EP-SPARQL) [7] propose a time-window based processing of Resource Descriptor Framework (RDF) streams for continuous SPARQL queries. On the other hand, the SPARQL Event Processing Architecture (SEPA) [8] proposes a different approach to Linked Data dynamics. The main goal of SEPA is to build a novel query notification mechanism on top of the standard SPARQL 1.1 protocol and languages. SEPA offers a service to receive streams of notifications about changes (i.e., added and removed bindings) in a SPARQL query result set [9], [10]. The software architecture of a SEPA broker is shown in Figure I.

Furthermore, thanks to the implemented publish-subscribe mechanism, SEPA enables an application design pattern where software agents can interact and synchronize to implement interoperable, distributed and context-aware Web applications. In order to scale at Web level, the performance of SEPA should be increased by implementing smarter algorithms for the subscriptions processing [11].

In this paper we propose an algorithm which enable the extraction of inserted and deleted quads (i.e., triples within a graph) from a SPARQL 1.1 Update [12]. This would allow to infer changes in a query result set (i.e., in the best case, by binding all the variables in the query) and to filter out the subscriptions that are not affected by the modification of the knowledge base.

The next section provides the background on which the algorithm described in Section III is based on. Eventually, we derive some considerations and plan future work in the conclusion.

Fig. 1. SEPA broker software architecture. It should be noted that a new update can processed only after all the previous notifications produced by the previous update have been sent (EOP call). The sequential update processing is granted by a FIFO queue.



II. Background

The baseline algorithm implemented by SEPA is listed in Algorithm 1. The algorithm executes all the registered SPARQL queries (i.e., subscriptions) and for each of them it compares the previous results with the current ones to retrieve the added and removed bindings. It is evident that with this algorithm, the SEPA service will not scale with respect to the number of active subscriptions and the size of knowledge base (i.e., number of query results).

The role played by the inserted and deleted quads on the subscription processing can be clarified by considering a simple example. Suppose to have two active subscriptions: S1 (see Listing 1) and S2 (see Listing 2).

```
PREFIX foaf : <http://xmlns.com/foaf/0.1/>
SELECT ?a
WHERE {
  GRAPH ?g {
    ?a foaf:familyName "Rossi".
  }
}
```

Listing 1. An example of a SPARQL subscription that is not effected by the update in *Listing 3*

ISSN 2305-7254

Algorithm 1 Naive algorithm to retrieve added and removed bindings in a set of SPARQL subscriptions. The process is executed on every update operation.

Input:
$Subscriptions$: list of the active subscriptions

for all $S \in Subscriptions$ **do**
  $oldResults \leftarrow getPrevious(S)$
  $subQuery \leftarrow getQuery(S)$
  $newResults \leftarrow$ submit $subQuery$ to the Endpoint
  $Inserted \leftarrow \emptyset$
  $Deleted \leftarrow \emptyset$
  for all $Q \in oldResults$ **do**
    if Q is not in $newResults$ **then**
      add Q to $Deleted$
    end if
  end for
  for all $Q \in newResults$ **do**
    if Q is not in $oldResults$ **then**
      add Q to $Inserted$
    end if
  end for
  send $Notification$ with $Inserted$ and $Deleted$
end for
return

```
PREFIX foaf : <http://xmlns.com/foaf/0.1/>
SELECT ?a ?name
WHERE {
  GRAPH ?g {
    ?a foaf:name ?name.
  }
}
```

Listing 2. An example of a SPARQL subscription that is effected by the update in *Listing 3*

Now a client performs the update in *Listing 3* which changes the name of every person named ”Pietro to ”Mario.

```
PREFIX foaf : <http://xmlns.com/foaf/0.1/>
DELETE {
  GRAPH ?g {
        ?a foaf:name "Pietro"
  }
}
INSERT {
  GRAPH ?g {
        ?a foaf:name "Mario".
  }
}
WHERE {
  GRAPH ?g {
    ?a foaf:name "Pietro".
  }
}
```

Listing 3. A SPARQL update which updates the name of every person named ”Pietro” to ”Mario”

While the query result set of S1 would not change as consequence of the update, the S2 subscription could bring to a notification if the results have changed. The inserted and removed quads would allow to filter out S1 and reduce the complexity of the S2 query (i.e., the ?name variable would be bind with ”Mario” and ”Pietro”).

A SPARQL update operation (see Listing 3) is always executed on a graph store. The store is composed by a set of *named graphs* and exactly one *default graph*. Each graph contains a set of RDF triples and can be named trough an IRI. Therefore the store is actually made up by a set of *quads*: $< graphIri, subject, predicate, object >$. This collection of elements contained in the store can be modified by SPARQL update primitives that are divided in two categories: *graph management* and *graph update*.

The *graph management* operations are those which modify the set of graphs in the store operating on the entire graph. For example, the *MOVE* operation moves the whole set of triples from graph A to graph B and the *DROP* operation deletes a named graph. Other graph management primitives can be found in [12].

The *graph update* operations, on the other hand, are focused on triple modification within graphs. Like for example *INSERT DATA* operation which can insert a specific s et of triples inside a graph or *INSERT/DELETE* which can update triples that satisfy a certain query. For further details see [12].

III. Algorithm

According to the SPARQL 1.1 Protocol [13], SPARQL endpoints are not required to provide a detailed result of a SPARQL Update operation. Most of them just report if it was successful or not. For example Blazegraph (https://www.blazegraph.com/) returns an HTML page with the timings and the status of the request. Nevertheless, this information could be useful for applications and services that use the endpoint as the main long term storage.

With reference to the SPARQL Event Processing Architecture (SEPA), as suggested by [14], the incremental changes of the RDF data set could potentially avoid querying the knowledge base. The algorithm here proposed is built on top of the SPARQL 1.1 Protocol and SPARQL 1.1 Update Language, and it is aimed to do not affect the underpinning SPARQL endpoint implementation, neither to propose any change in the above mentioned protocol and language. A naive algorithm could just query the whole knowledge base before and after an update, and then compares the results. Of course this solution is not feasible due to the number of quads that are typically stored inside a SPARQL endpoint (e.g., millions or trillions). Moreover, graph management primitives and *INSERT/DELETE* operations do not declare a list of quads to be modified, but those quads are actually computed at run time by the SPARQL endpoint.

The algorithm here proposed exploits the SPARQL *CONSTRUCT* primitive to retrieve a subset of triples that are contained within a graph of a RDF graph store. For example, with reference to the update in Listing 3, the triples that would possibly be effected by the update can be obtained with two construct queries. In Listing 4 is shown the *CONSTRUCT* that could be used to create the RDF triples that would be deleted. The same *CONSTRUCT* can be used to obtain the inserted

triples by replacing, within the *CONSTRUCT* {...} statement, ”Pietro” with ”Mario”.

```
CONSTRUCT {
    ?a foaf:name "Pietro"
  }
}
WHERE {
  GRAPH ?g {
    ?a foaf:name "Pietro".
  }
}
```

Listing 4. The *CONSTRUCT* query to create the triples that would be deleted according to the update in Listing 3.

Are these triples really inserted or removed? In fact, a triple will be really inserted if it is not present in the RDF store, while a triple will be really deleted if it is present in the RDF store. In the first instance, the presence of a triple in a RDF store can be asserted using the *ASK* query statement.

The algorithm we proposed to extract the actually inserted and deleted quads is detailed in Algorithm 3 and it makes use of the following functions (i.e., functions in bold require accessing the SPARQL endpoint):

- *isGraphManagement*($update$): returns true if the input update is one of the graph management SPARQL operations (i.e., *MOVE*, *COPY*, *ADD*, *CLEAR*, *CREATE*, *DROP*, *LOAD*).
- *isDataUpdate*($update$): returns true if the input update is an *INSERT/DELETE DATA* operation.
- *extractInsQuads*($update$): extracts the quads defined in the *INSERT DATA* statement. In this process declared triples are combined with the default graph IRI while the quads are extracted as they are.
- *extractDelQuads*($update$): extracts the quads defined in the *DELETE DATA* statement. The extraction process is similar to *extractInsQuads*.
- *hasGraphVariables*($update$): checks the presence of graph variables inside the *INSERT* or *DELETE* statements. The function returns true if at least a graph variable is found, false otherwise.
- *combine*($triples$,$graph$): transforms a list of triples into a list of quads combining them with the input $graph$.
- *buildUpdate(*$iQuads$,$dQuads$*)*: creates a SPARQL *INSERT/DELETE DATA* operation from two lists of inserted (i.e., $iQuads$) and deleted (i.e., $dQuads$) quads.
- *getTargetGraph(*$update$*)*: returns the target graph of a graph management update operation.
- *getSourceGraph(*$update$*)*: returns the source graph of a graph management update operation.
- **getGraphs**($update$): executes a SPARQL *SELECT* query selecting only graph variables defined inside the *INSERT* or *DELETE* statement of a SPARQL *INSERT/DELETE* primitive. It is used to retrieve a list of graph *IRIs*.
- **getInsTriples**($graph$,$update$): builds and executes a SPARQL *CONSTRUCT* query. The query retrieves all the triples effected by the *INSERT* statement of the $update$ in the specified $graph$.
- **getDelTriples**($graph$,$update$): builds and executes a SPARQL *CONSTRUCT* query. The query retrieves all the triples affected by the *DELETE* statement of the $update$ in the specified $graph$.
- **getTriples**($graph$): builds and executes a SPARQL *CONSTRUCT* query. The query retrieves all the triples affected inside the $graph$ specified as input.
- **isPresent**($quad$): uses a *ASK* query to check the presence of a quad inside the data set. It returns true if the quad is present, false otherwise.
- **transform**($update$): transforms a graph management update to a *INSERT/DELETE DATA* update primitive implementing the Algorithm 2.

Since different SPARQL update operations have different semantics, the first step of the algorithm verifies the type of operation. On line 19, the algorithm checks if the update is from the graph management family using the *isGraphManagement* function. If yes, the update is converted into an *INSERT DATA/DELETE DATA* primitive thanks to the **transform** function. As shown in Algorithm 2, the conversion is done by retrieving the triples using the **getTriples** function and combining them with the corresponding graph using the *combine* function.

Algorithm 2 Transform a graph management operation into a INSERT/DELETE DATA primitive

1: **Input:**
2: $Update$: a SPARQL graph management primitive (i.e., MOVE, COPY, ADD, CLEAR, CREATE, DROP, LOAD)
3: **Output:**
4: Ret : a SPARQL INSERT/DELETE DATA primitive
5:
6: **Temp:**
7: $Graph$, $InsGraph$, $DelGraph$: IRI of graphs
8: $InsTriples$, $DelTriples$: sets of RDF triples
9: $InsQuads$, $DelQuads$: sets of RDF quads
10:
11: $InsTriples \leftarrow \emptyset$
12: $DelTriples \leftarrow \emptyset$
13: $InsGraph \leftarrow \emptyset$
14: $DelGraph \leftarrow \emptyset$
15:
16: **switch** ($Update$)
17: **case** $CREATE$**:**
18: $Ret \leftarrow buildUpdate(\emptyset, \emptyset)$
19: **return** Ret
20: **case** $DROP$**:**
21: $DelGraph \leftarrow getSourceGraph(Update)$
22: $DelTriples \leftarrow \mathbf{getTriples}(Graph)$
23: **case** $MOVE$**:**
24: $InsGraph \leftarrow$ *getTargetGraph(*$Update$*)*
25: $InsTriples \leftarrow \mathbf{getTriples}(InsGraph)$
26: $DelGraph \leftarrow$ *getSourceGraph(*$Update$*)*
27: $DelTriples \leftarrow \mathbf{getTriples}(DelGraph)$

28: **case** $COPY$**:**
29: $InsGraph \leftarrow getTargetGraph(Update)$
30: $Graph \leftarrow getSourceGraph(Update)$
31: $InsTriples \leftarrow \mathbf{getTriples}(Graph)$
32: $DelGraph \leftarrow InsGraph$
33: $DelTriples \leftarrow \mathbf{getTriples}(DelGraph)$
34: **case** ADD**:**
35: $InsGraph \leftarrow getTargetGraph(Update)$
36: $Graph \leftarrow getSourceGraph(Update)$
37: $InsTriples \leftarrow \mathbf{getTriples}(Graph)$
38: **case** $CLEAR$**:**
39: $DelGraph \leftarrow getTargetGraph(Update)$
40: $DelTriples \leftarrow \mathbf{getTriples}(DelGraph)$
41: **case** $LOAD$**:**
42: $InsTriples \leftarrow$ HTTP request to document URI
43: $InsGraph \leftarrow getTargetGraph(Update)$
44: **end switch**
45:
46: $InsQuads \leftarrow combine(InsTriples, InsGraph)$
47: $DelQuads \leftarrow combine(DelTriples, DelGraph)$
48: $Ret \leftarrow buildUpdate(InsQuads, DelQuads)$
49:
50: **return** Ret

Then, on line 23, the update type is checked again using the *isDataUpdate* function. Notice that at this point the operation can be only a graph update primitive. If the update is an *INSERT/DELETE DATA* then the algorithm extracts the quads defined inside the statements and inserts them into the lists of candidate's quads to be inserted and/or deleted (i.e., *IQuadsTemp, DQuadsTemp*).

Otherwise some extra steps are required to obtain the candidate quads. In fact, a SPARQL *INSERT/DELETE* primitive could include variables inside the *INSERT* or *DELETE* clause. The bindings of these variables will be retrieved according to the *WHERE* clause. So, the definition of the *CONSTRUCT* primitive is not straightforward as in the *INSERT/DELETE DATA* case. For example, the update in Listing *3* updates the name of all the people named "Pietro" to "Mario" in every named graph defined in the graph store. But at compile time, the IRIs of the effected graphs are unknown. As the CONSTRUCT operation cannot contain graph variables (https://www.w3.org/TR/sparql11-query/#construct), the IRIs of the graphs effected by the update must be retrieved before building the CONSTRUCT. This is done by the **getGraphs** function and an example of the query used to retrieve such IRIs is shown in Listing 5.

```
PREFIX foaf : <http://xmlns.com/foaf/0.1/>
SELECT ?g
WHERE {
  GRAPH ?g {
    ?a foaf:name "Pietro".
  }
}
```

Listing 5. A SPARQL SELECT query which retrieves the graphs effected by the update in Listing 3

Algorithm 3 Quads extraction from a generic SPARQL 1.1 update operation, including graph management operations

1: **Input:**
2: $Update$: the SPARQL 1.1 Update
3:
4: **Output:**
5: $IQuads$: set of quads that are really inserted (i.e., they were not present in the store)
6: $DQuads$: set of quads that are really removed (i.e., they were present in the store)
7:
8: **Temp:**
9: $IQuadsTemp$: set of quads that could be inserted (i.e., they could be already present in the store)
10: $DQuadsTemp$: set of quads that could be removed (i.e., they could be not present in the store)
11: $graphs$: set of graphs URIs
12: $triples$: set of RDF triples
13:
14: $IQuads \leftarrow \emptyset$
15: $DQuads \leftarrow \emptyset$
16: $IQuadsTemp \leftarrow \emptyset$
17: $DQuadsTemp \leftarrow \emptyset$
18:
19: **if** *isGraphManagement(*$Update$*)* **then**
20: $Update \leftarrow$ **transform(**$Update$**)**
21: **end if**
22:
23: **if** *isDataUpdate(*$Update$*)* **then**
24: $IQuadsTemp \leftarrow$ *extractInsQuads(*$Update$*)*
25: $DQuadsTemp \leftarrow$ *extractDelQuads(*$Update$*)*
26: **else**
27: $graphs \leftarrow$ "$default$"
28: **if** *hasGraphVariables(*$Update$*)* **then**
29: $graphs \leftarrow \mathbf{getGraphs}(Update)$
30: **end if**
31:
32: **for all** $G \in graphs$ **do**
33: $triples \leftarrow \mathbf{getInsTriples}(G, update)$
34: $IQuadsTemp \leftarrow combine(triples, G)$
35: $triples \leftarrow \mathbf{getDelTriples}(G, update)$
36: $DQuadsTemp \leftarrow combine(triples, G)$
37: **end for**
38: **end if**
39:
40: **for all** $IQ \in IQuadsTemp$ **do**
41: **if isPresent**(IQ) **then**
42: Add IQ to $IQuads$
43: **end if**
44: **end for**
45:
46: **for all** $DQ \in DQuadsTemp$ **do**
47: **if not isPresent**(DQ) **then**
48: Add DQ to $DQuads$
49: **end if**
50: **end for**
51:
52: **return** $IQuads$, $DQuads$

In order to build the list of quads that are candidate to be inserted and/or deleted, the algorithm follows a procedure (see lines 32-37) similar to the one implemented by the Algorithm 2. Thanks to the **getInsTriples**, **getDelTriples** and *combine* functions, the algorithm first extracts the triples from each graph and then combines these triples with the corresponding graph IRI.

Eventually, from line 40 to line 50, thanks to the **isPresent** function, the algorithm checks the actual existence (or not existence) of the quads into the quad store. Relevant quads are then inserted into the lists of quads to be returned by the algorithm (i.e., *IQuads, DQuads*).

IV. Conclusion and Future work

This article presents an algorithm that can be implemented to extract the quads that are really inserted and deleted by a generic SPARQL 1.1 Update operation, without effecting the underpinning SPARQL 1.1 Protocol service implementation. The algorithm will be implemented and evaluated as part of the SPARQL Event Processing Architecture. This would lead to understand if the overhead introduced by the processing of a SPARQL update can be repaid by reducing the subscription computational cost in terms of number of subscriptions to be evaluated per update and processing time of a single subscription.

Acknowledgment

We would like to thank the organizers of the 23nd FRUCT Conference for asking us to submit this short paper which provides more details on the poster we presented at the conference.

References

[1] C. Bizer, T. Heath, and T. Berners-Lee, "Linked Data - The Story So Far," *International Journal on Semantic Web and Information Systems*, vol. 5, no. 3, pp. 1–22, 2009.

[2] T. Käfer, A. Abdelrahman, J. Umbrich, P. O'Byrne, and A. Hogan, "Observing linked data dynamics," in *The Semantic Web: Semantics and Big Data* (P. Cimiano, O. Corcho, V. Presutti, L. Hollink, and S. Rudolph, eds.), (Berlin, Heidelberg), pp. 213–227, Springer Berlin Heidelberg, 2013.

[3] B. Norton and R. Krummenacher, "Consuming Dynamic Linked Data," in *COLD*, 2010.

[4] J. Umbrich, B. Villazón-Terrazas, and M. Hausenblas, "Dataset Dynamics Compendium: A Comparative Study," in *Proceedings of the First International Workshop on Consuming Linked Data, Shanghai, China, November 8, 2010*, 2010.

[5] D. F. Barbieri, D. Braga, S. Ceri, and M. Grossniklaus, "An Execution Environment for C-SPARQL Queries," in *Proceedings of the 13th International Conference on Extending Database Technology*, EDBT '10, (New York, NY, USA), pp. 441–452, ACM, 2010.

[6] J.-P. Calbimonte, O. Corcho, and A. J. G. Gray, "Enabling Ontology-Based Access to Streaming Data Sources," in *The Semantic Web – ISWC 2010* (P. F. Patel-Schneider, Y. Pan, P. Hitzler, P. Mika, L. Zhang, J. Z. Pan, I. Horrocks, and B. Glimm, eds.), (Berlin, Heidelberg), pp. 96–111, Springer Berlin Heidelberg, 2010.

[7] D. Anicic, P. Fodor, S. Rudolph, and N. Stojanovic, "EP-SPARQL: A Unified Language for Event Processing and Stream Reasoning," in *Proceedings of the 20th International Conference on World Wide Web*, WWW '11, (New York, NY, USA), pp. 635–644, ACM, 2011.

[8] L. Roffia, P. Azzoni, C. Aguzzi, F. Viola, F. Antoniazzi, and T. Salmon Cinotti, "Dynamic Linked Data: A SPARQL Event Processing Architecture," *Future Internet*, vol. 10, no. 4, p. 36, 2018.

[9] C. Aguzzi, F. Antoniazzi, F. Viola, and L. Roffia, "SPARQL 1.1 Subscribe Language," 2018. [Online; accessed 21-November-2018].

[10] A. Seaborne, "SPARQL 1.1 Query Results JSON Format." Web: https://www.w3.org/TR/sparql11-results-json/, 2013. [Online; accessed 21-November-2018].

[11] L. Roffia, F. Morandi, J. Kiljander, A. D'Elia, F. Vergari, F. Viola, L. Bononi, and T. S. Cinotti, "A Semantic Publish-Subscribe Architecture for the Internet of Things," *IEEE Internet of Things Journal*, dec 2016.

[12] P. Gearon, A. Passant, and A. Polleres, "SPARQL 1.1 Update." Web: https://www.w3.org/TR/sparql11-update/, 2013. [Online; accessed 21-November-2018].

[13] L. Feigenbaum, G. T. Williams, K. G. Clark, and E. Torres, "SPARQL 1.1 Protocol." Web: https://www.w3.org/TR/sparql11-protocol/, 2013. [Online; accessed 21-November-2018].

[14] F. Schmedding, "Incremental SPARQL evaluation for query answering on linked data," in *Proceedings of the Second International Conference on Consuming Linked Data-Volume 782*, pp. 49–60, Citeseer, 2011.