De-icing federated SPARQL pipelines: a method for assessing the "freshness" of result sets

Damien Graux[1,2(✉)], Fabrizio Orlandi[2(✉)], and Declan O'Sullivan[2]

[1] Inria, Université Côte d'Azur, CNRS, I3S, France
[2] ADAPT SFI Centre, Trinity College Dublin, Ireland
{grauxd,orlandif,declan.osullivan}@tcd.ie

Abstract. In recent years, the ever-increasing number of available linked-data endpoints has allowed the creation of complex data pipelines leveraging these massive amounts of information. One crucial challenge for federated pipeline designers is to know when to query the various sources they use in order to obtain fresher final results. In other words, they want to know when a data update on a specific source impacts their own final results. Unfortunately, the SPARQL standard does not provide them with a method to be aware of such updates; and therefore pipelines are regularly relaunched from scratch, often uselessly.
To help them decide when to get fresher results, we propose a constructive method. Practically, it relies on digitally signing result sets from federated endpoints in order to create a specific query able to warn when, and explain why, the pipeline result set is outdated. In addition, as our solution is exclusively based on SPARQL 1.1 built-in functions, it is fully-compliant with all the endpoints.

1 Introduction

During the past decades, the number of linked open datasets has rapidly increased[1]. These datasets are structured following the W3C standard Resource Description Framework (RDF) [6] and share knowledge on various domains, from the generalist ones such as DBpedia [1] or WikiData [7] to the most specialised ones, *e.g.* SemanGit [5]. This abundance of open datasets and SPARQL [2] endpoints led not only researchers but also businesses to integrate RDF graphs into their complex data pipelines. In particular, businesses are increasingly leveraging Semantic Web technologies to structure their own data and create value, sometimes integrating external Linked Data to enrich their analyses [4].

Benefiting from these two decades of developments made by the community, it is now possible to deploy Semantic Web pipelines and to adapt them to a wide range of needs and use-cases. Recent developments have been, for example, focused on distributed systems or on connecting Semantic Web data management systems together with non-RDF centric systems, paving the road to querying

[1] From 2010 to 2020, the LOD-cloud has grown from 203 to 1 255 datasets, approximately: https://lod-cloud.net/

heterogeneous data. As a consequence of this increasing complexity of the use-cases, the pipelines themselves are getting more complicated, and often rely on several distinct data sources using SPARQL federation techniques, in order to compute their final results.

Hence, as data available may change, these pipelines (or parts of them) are frequently re-run in order to get fresher results. However, lots of times they are re-run unnecessarily as datasets have not been updated in the meantime in ways that impact the result sets of the pipeline. All these operations are leading to a waste of computation power and loads on the network.

In this article, mainly dedicated to SPARQL practitioners and data pipeline designers, we present some possibilities provided by SPARQL 1.1 [2] to hash query result sets with a focus on the case of SPARQL federation of sources with the `SERVICE` keyword. In particular, we will discuss how these methods can be used to optimise data pipelines avoiding expensive re-computation of results when data triples have not been updated.

2 Hashing features in SPARQL

In this Section, we review several use-cases where the use of SPARQL hashing features becomes relevant. In particular, we focus on various methods to obtain a "signature" of a SPARQL result set.

2.1 SPARQL 1.1 capabilities

The SPARQL standard provides a large set of built-in functions, from ones dedicated to strings to specific ones about dates. These can be used by query designers to refine their result set. In particular, the standard offers a set of five hash functions[2]: MD5, SHA1, SHA256, SHA384 & SHA512.

General signature of the hash functions:

```
simple literal  hash_function (simple literal arg)
simple literal  hash_function (xsd:string arg)
```

Example using MD5:

```
H = md5("ab") =  md5("ab"^^xsd:string)
H = "187ef4436122d1cc2f40dc2b92f0eba0"
```

These functions accept either RDF literals or strings as argument and return the hash as a literal. In addition, a `xsd:string` or its corresponding literal should return the same result. In the 'MD5' example above, the hash value represents the result of a simple SPARQL query[3].

2.2 From hashing an RDF graph ...

Technically, the standard hash functions provide the query designer with useful tools to, for instance, check/sign a specific value within a `filter` field. Nevertheless, they could also be used for more complex tasks that simply checking a

[2] https://www.w3.org/TR/sparql11-query/#func-hash

[3] `select * where{ values ?x {"ab" "ab"^^xsd:string} bind (md5(?x) as ?H)}`

value. For example, in a more extreme case, they could be used over a complete RDF graph through a SPARQL endpoint: one can extract all the triples available with `select * where {?s ?p ?o}`, and then hash all of them, aggregated with a `group_concat` function. This could look like so:

```
SELECT (SHA1(GROUP_CONCAT(?tripleStr ; separator=' \n'))) AS ?nTriples
WHERE { ?s  ?p  ?o
  BIND(CONCAT(STR(?s), " ", STR(?p), " ", STR(?o)) AS ?tripleStr) }
```

In the previous query, the triples `?s ?p ?o` are cast by element to a string (`STR`), and then concatenated to form a "triple". The recomposed list of triples is then grouped into one single string (`GROUP_CCONCAT`) and finally hashed.

Obviously, this "naive" approach has drawbacks. *First*, the result depends on the order of the triples returned by the triplestore: a workaround can be achieved adding *e.g.* `ORDER BY ?s ?p ?o datatype(?o) lcase(lang(?o))`. *Second*, this method has a scalability issue, as all the graph is loaded in-memory before the hash call. We therefore recommend this approach to sign small RDF graphs, *e.g.* ontologies or small result sets. *Finally*, this method does not address the complex case of blank node identification as *e.g.* {_:a p o} and {_:b p o} do not have the same hashes (see *e.g.* [3] for algorithmic solutions).

2.3 ... To hashing SPARQL result sets

More pragmatically however, these hash functions can be used in order to sign SPARQL query result sets, typically, they allow users to compare different query results for the same SPARQL endpoint. As we know, on the same endpoint, the same query (without calls to functions like `RAND` or `NOW`) is supposed to return the same result set for the same dataset. A hash of the results could be computed by the endpoint and be compared with a previously obtained one. In case of a mismatch, the query (and the rest of the pipeline) could be run again. Assuming `Q` is the considered SPARQL `select` query, we propose the following steps to generate the query which computes the hash of the results of `Q`:

1. Extract and sort the list of distinguished variables `V` (if a `*` is given, the considered variables are the ones involved in the `where`);
2. Wrap `Q` in a `select *` query ordered by `V`;
3. Embed the obtained query in a `select` query computing the hash of the grouped concatenation of the cast (to string) distinguished variables.

To give an example, if we consider the query[4] which extracts from DBpedia the current members of English-named Punk_rock groups, `Q`=

```
SELECT ?members ?bandName WHERE {
 ?band dbo:genre dbr:Punk_rock . ?band dbp:currentMembers ?members .
 ?band foaf:name ?bandName FILTER(langMatches(lang(?bandName), "en")) }
```

[4] The queries can be tested directly on DBpedia: the query `Q` and its `Q_hash`. As of April 12th 2021, `Q_hash` returns `?H = "967d2c8c0a82038d8478d476fa41e14f"`.

Its sorted list of distinguished variables would be `?bandName ?members`. And to obtain a (MD5-)hash of the results of `Q`, we should run:

```
SELECT MD5(GROUP_CONCAT(CONCAT(STR(?bandName),STR(?members)); separator=' \n'))
as ?H WHERE {
 SELECT * WHERE {                        # Collecting all the ordered results
  SELECT ?members ?bandName WHERE {      # The original query
   ?band dbo:genre dbr:Punk_rock . ?band dbp:currentMembers ?members.
   ?band foaf:name ?bandName FILTER(langMatches(lang(?bandName), "en")) }
 } ORDER BY ?bandName ?members           # Ordering by distinguished variables
}
```

Using this method, a SPARQL pipeline designer can easily obtain a hash corresponding to a result set, and store it for computing a comparison later. She would therefore know if RDF data updates have been impacting, in the meanwhile, her results. On advantage of such a method lays in the place where the computations happens: the SPARQL endpoint which could thereby be in charge of having otpimisation techniques (such as result caching or sub-view updating) to retrieve the result sets faster.

2.4 Gaining in precision with the `SERVICE` keyword

SPARQL pipelines often rely on several data sources that are queried on-the-fly and whose sub-results are aggregated to form the final result set. Practically, the SPARQL standard provides the designers with a built-in function to call an external endpoint with specific conditions: the `SERVICE` keyword. This keyword is supposed to take as arguments a SPARQL endpoint URI[5] and a set of conditions similarly to the `where` clause.

Consequently, by the mean of SPARQL sub-queries it is possible to refine the previous method when the designer sets up federated queries. Indeed, each `SERVICE` call can be extracted and encapsulated following the steps presented previously. For example, considering `SELECT ?s WHERE { ?s ?p ?o . SERVICE <URI> { ?s ?t ?u . }}`, we can obtain a finer precision computing the following two hashing SPARQL queries:

```
SELECT (MD5(GROUP_CONCAT(CONCAT(
  STR(?s),STR(?t),STR(?u));
  separator=' \n')) as ?ServiceHash)
WHERE{
 SELECT ?s ?t ?u WHERE {
  SERVICE <URI> { ?s ?t ?u. }
 } ORDER BY ?s ?t ?u
}
```

```
SELECT (MD5(GROUP_CONCAT(CONCAT(
  STR(?s)); separator=' \n'))
  as ?ResultSetHash )
WHERE{
 SELECT * WHERE {
  SELECT ?s WHERE {
   ?s ?p ?o. SERVICE <URI>
   { ?s ?t ?u. } } } ORDER BY ?s }
```

The first query (left-hand side) focuses on computing a hash of the results set

[5] According to the SPARQL 1.1 standard, this SPARQL endpoint URI may also be a variable, *e.g.* "`SERVICE ?url { ... }`", requiring the endpoint to first retrieve the possible entities for this variable and then call the external endpoints applying the set of conditions. However, this case is currently out of our scope.

obtained from the external source; and the second one hashes the complete result set obtained once the local and the external results are joined.

More generally, one needs to generate one dedicated hashing query per `SERVICE` call plus a general hashing query for the main/top-level SPARQL query. By using this strategy, the query designer is then able to create her merkle tree[6] of her different sources' sub-results. Moreover, this method is fully compliant with the SPARQL standard as it uses exclusively built-in functions. Finally, this "incremental" approach allows the users to know which source has been updated in a way that might impact the general query.

3 Automation for complex federated SPARQL queries

The aforedescribed steps to generate the queries that obtain the hashes, splitting the query into pieces based on `SERVICE`, are easy to automate and allow users to know when to relaunch their pipelines. In order to help pipeline designers in practice, we serve our method at: https://dgraux.github.io/SPARQL-hash/ where our query converters can be used directly by developers to generate queries computing the hash of their result sets both for "regular" SPARQL queries and federated ones.

In the case of source federation, as the generation of several hashing queries (as presented in the previous section) is tedious and prone to committing errors, we also provide a generator dedicated to federated queries. Practically, our generator takes the SPARQL query `Q` and returns a new one `Q_hash` which take care of computing all the necessary hashing sub-queries once run on the designers' SPARQL endpoint[7]. Specifically, `Q_hash`'s result is another SPARQL query `Q_check` which is in charge of running[8] the various hashing sub-queries in order to compare the hashes with the previous ones obtained by `Q_hash`. By construction, `Q_hash` is supposed to be run only once and `Q_check` should be the one -once obtained- used regularly to check if `Q` needs to be relaunched in order to update the designer's final result set. Furthermore, `Q_check` is built in a way which allows the query designers to know why she should update her final result set; it actually explains which specific source is updated.

As `Q_check` and more `Q_hash` tend to be long, we refer the reader to the extensive set of examples on our website for details. Notwithstanding, for the sake of clarity, let's consider the following simplified query (`Q`): `select * where { ... service uri {...}}`. Let's now name Q_{serv} and Q_{gene} the sub-queries generating `"123"` and `"abc"`: the hashes (following the methods proposed in Sections 2.3 & 2.4) for the `service` part and for `Q` respectively. Therefore, `Q_check` would have the following sketch:

```
select ?res where {
 { Q_serv } # To compute a fresh hash for the service
```

[6] Following recursively the `SERVICE` calls.
[7] *i.e.* the same endpoint as the one where `Q` is supposed to be run.
[8] Like `Q_hash`, `Q_check` should be run on the endpoint where `Q` is supposed to be run.

```
 { Q_gene } # To compute a fresh hash for Q
 if (Q_serv="123"){
   if (Q_gene="abc") {?res="No update"}
   else {?res="Local data has changed"}
 } else {
   ?res="External source has changed"
 }
}
```

As presented above, `Q_check` inspects the possible updates of hash results using a set of nested conditions. In particular, this set of conditions follows the exploration of a merkle tree starting from the bottom layer. Therefore, by structuring the query like so, an optimised SPARQL endpoint might be able to properly order its query plan in order to evaluate sequentially the various sub-queries and stop as soon as possible, saving computational resources.

4 Conclusions

This article describes how to improve existing Semantic Web data pipelines with a SPARQL-based method that helps in identifying when query results have changed. In particular, it sheds lights on a possible strategy when dealing with complex federated queries. It allows to re-run pipelines only when interesting parts of the original datasets have been updated. By using SPARQL to compute the signature of the query results, it avoids large result sets to be sent over the network. Finally, it also helps the designers identify what are the external sources that have been updated. We hope this will inspire developers to use the hash functions provided by the standard.

References

1. Auer, S., Bizer, C., Kobilarov, G., Lehmann, J., Cyganiak, R., Ives, Z.: DBpedia: A nucleus for a web of open data. In: The semantic web. Springer (2007)
2. Harris, S., Seaborne, A., Prud'hommeaux, E.: Sparql 1.1 query language. W3C recommendation **21**(10), 778 (2013)
3. Hogan, A.: Canonical forms for isomorphic and equivalent RDF graphs: algorithms for leaning and labelling blank nodes. ACM Transactions on the Web (TWEB) **11**(4), 1–62 (2017)
4. Junior, A.C., Orlandi, F., Graux, D., Hossari, M., O'Sullivan, D., Hartz, C., Dirschl, C.: Knowledge graph-based legal search over german court cases. In: The Semantic Web: ESWC 2020 Satellite Events. LNCS, vol. 12124, pp. 293–297. Springer (2020)
5. Kubitza, D.O., Böckmann, M., Graux, D.: SemanGit: A linked dataset from git. In: International Semantic Web Conference. pp. 215–228. Springer (2019)
6. Manola, F., Miller, E., McBride, B., et al.: RDF primer. W3C recommendation **10**(1-107), 6 (2004)
7. Vrandečić, D., Krötzsch, M.: Wikidata: a free collaborative knowledgebase. Communications of the ACM **57**(10), 78–85 (2014)