SPARQLe Up Your Knowledge Graphs with on-the-fly Computed Triples

Geert Vanderhulst, Johan Van Noten and Davy Maes

CodesignS, Flanders Make, Leuven, Belgium

Abstract

Knowledge graphs (KGs) provide organisations with flexibility and insights hard to achieve with conventional databases. However, not every type of data that lives inside an organisation integrates well in a KG (e.g. time-series data, log files, unmaterialised data computed by scripts, etc). As a result, data scientists still need to combine data from different sources and master different query languages to answer global questions. In this paper, we introduce *virtual predicates* as a solution to query data via SPARQL that is purposely not part of a KG. We augment existing SPARQL endpoints with triples that are generated on the fly based on the query context. Our solution consists of a SPARQL-OTFC proxy endpoint through which queries are routed and forwarded to a myriad of data sources to collect and compute missing data.

1. Introduction

A knowledge graph (KG) could be considered as a single entry point for data scientists and visualisation tools to query and inspect an organisation's data. In reality, however, there are several barriers to overcome: some data types such as time-series data do not scale well to triple stores, privacy and security policies may require sensitive data such as credit card details to be stored separately and on-demand computations (e.g. embedded in stored procedures, scripts, ...) can be difficult to integrate into SPARQL extensions. We assume a KG and its ontology describing an organisation's core data concepts and relationships, co-exists with other heterogeneous data sources. Our goal is to make *any relevant data* – stored or computed, part of the KG or not – accessible via a uniform SPARQL interface. To this end, we extend data in a KG with triples that are generated on the fly based on the query asked. We introduce *virtual predicates*, imitating regular predicates, to transparently query triples that do not exist (and fit) in the KG. These virtual predicates enable us to:

- Dynamically pull in selective data from heterogeneous data sources such as specialty databases, file servers and REST APIs;
- Perform advanced calculations and generate arbitrary datatypes on the fly;
- Regulate access to sensitive data based on user and virtual predicate policies.

We illustrate our approach via a proof-of-concept implementation that combines data from publicly available data sources.

```
1  SELECT ?mcdo WHERE {
2    ?mcdo osmt:brand "McDonald's";
3          osmt:addr:city "New York";
4          osmt:addr:street ?street;
5          otfc:iceCreamBroken false.
6    FILTER (?street = "6th Avenue" ||
7            ?street = "Broadway")
8  }
```

(a) user query

SPARQL-OTFC proxy

(1) compute   **(2) federate**

+

iceCreamBroken triples   OSM KG *sophox.org/sparql*

results

```
1  SELECT ?mcdo ?coords WHERE {
2    ?mcdo osmt:brand "McDonald's";
3          osmm:loc ?coords.
4  }
```

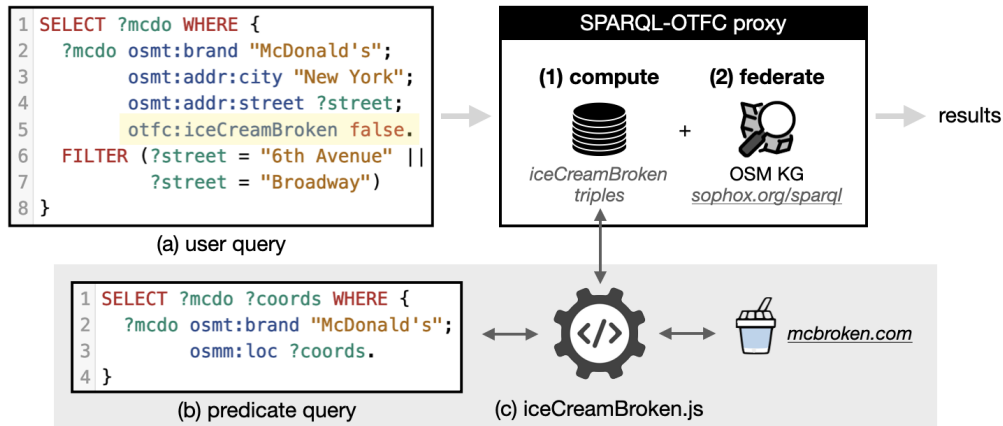(b) predicate query   (c) iceCreamBroken.js   *mcbroken.com*

Figure 1: Passing a user query with a virtual predicate through a SPARQL-OTFC proxy. In a compute step, a script collects data and generates triples for each virtual predicate. In a federate step, computed triples and data in a target KG are combined to answer the user query.

2. Related work

Virtual KGs are not new, with e.g. Ontop exposing non-semantic databases as SPARQL endpoints [6]. After creating mappings between an ontology and a relational database schema, a SPARQL query can be asked that is translated into SQL and SQL results are then converted back into SPARQL bindings. We advance this virtualisation effort by augmenting a KG (virtual or regular) with predicates that compute data that does not exist in its underlying data store. In [3], SPARQL queries are extended with special syntax that is rewritten into VOLT procedures when passed through a SPARQL proxy. Similar to SPARQL extension functions, GraphDB's magic predicates [1] and Jena's property functions [2], these approaches combine query rewriting with functions that accept query variables as arguments to compute data within a query's result set. Opposed to this, virtual predicates inject computed triples in a virtual KG prior to query execution and are syntactically equivalent to regular predicates by design.

3. Virtual predicates

Consider the user query in fig. 1(a) targeted at OpenStreetMap (OSM) [4], that fetches McDonald's restaurants on 6th Avenue and Broadway where the ice cream machine is currently working. Note that information about the state of a restaurant's ice cream machine is not contained in the OSM KG. It is dynamically generated from the highlighted `otfc:iceCreamBroken` virtual predicate via a script (fig. 1(c)) that collets data required for computation via a predicate query (fig. 1(b)) and ice cream machine states from mcbroken.com. This script is registered at the SPARQL-OTFC proxy through which the query is routed, using the corresponding virtual predicate's IRI as identifier. The proxy is built using Node.js and Comunica [5] and supports both JavaScript and Python interfaces for pluggable predicate scripts.

When a query is processed by SPARQL-OTFC, virtual predicates are resolved first via their IRI. Preprocessed queries are then forwarded by the proxy to target SPARQL endpoints such as

OSM Sophox. The two main steps in its processing pipeline – computation and federation – are discussed in section 3.1 and 3.2 respectively.

3.1. Computation

Just like a regular predicate, a virtual predicate P is bound to a subject variable (or blank node). For each matching subject S_i in the KG, a triple $\{S_i, P, O_i\}$ will be generated by a script implementing the virtual predicate, with O_i a computed literal or IRI. Consider as an example of such script the implementation of the `otfc:iceCreamBroken` virtual predicate in list. 1.

```
1   import Predicate from '../predicate.js';
2
3   export default class IceCreamBrokenPredicate extends Predicate {
4     static iri = 'http://flandersmake.be/otfc/iceCreamBroken'; // see user query, fig. 1(a)
5
6     static async compute(query, context, engine) {
7       let q = this.read('./queries/iceCreamBroken.sparql'); // see predicate query, fig. 1(b)
8       q = this.merge(q, query, 'mcdo', IceCreamBrokenPredicate.iri); // merge constraints
9       let triples = [];
10      await engine.run(q, context, data => { // generate a triple for each relevant mcdo
11        let broken = getStateFromMcBrokenDotCom(data.coords);
12        triples.push({
13          s: Predicate.iri(data.mcdo),
14          p: Predicate.iri(IceCreamBrokenPredicate.iri),
15          o: broken
16        });
17      });
18      return triples;
19    }
20  }
```

Listing 1: Extract from iceCreamBroken.js. Information from the OSG KG (coordinates of McDonald's restaurants) is used to obtain ice cream machine states from an external data source (mcbroken.com). From this data triples are generated that serve as input to a federation step (section 3.2).

Within the `compute` function, the script obtains S_i candidates (`?mcdo` instances) along with additional information that is not necessarily requested by the end user in what we call a predicate query. For instance, while a user may not ask for a restaurant's GPS coordinates, the predicate script still needs those to obtain the ice cream machine's state at that location. As the creator of a predicate query cannot predict how a virtual predicate will be used in user queries, she/he should collect *all* subject candidates from the KG and compute triples for each of them. However, since a user query will likely constrain the subject, the predicate query would overfetch and more triples than strictly needed will be computed. To avoid this, we extend the predicate query with constraints on P's subject (mcdo in our example) found in the user query at runtime. These include triple patterns from the basic graph pattern in which P appears along with any FILTER expressions that further constrain the variables in those patterns. For example, the predicate query in fig. 1(b) by default asks for all McDonald's branded OSM objects and their coordinates. However, the user query only asks for restaurants located in New York on Broadway or 6th Avenue. Lines 3, 4, 6 and 7 in the user query (fig. 1(a)) define these constraints

and are automatically merged into the predicate query which will now only return the set of strictly necessary results. Coordinates are then matched with JSON data obtained from mcbroken.com to acquire the state of the ice cream machines. This state is encoded as a simple boolean in triples such as `{ osmnode:2443892135, otfc:iceCreamBroken, false }` which are added to a "disposable KG" (i.e. a temporary triple store that caches results specific to a user query). All the information to answer the user query is now available, albeit distributed amongst the OSM KG and the disposable KG.

3.2. Federation

Once the disposable KG has been filled with computed data triples, the original user query is federated against the original KG and the disposable KG. We consider three approaches for federation:

1. *Automated federation*: the query engine decides how to probe the KGs and collect relevant data from either of them;
2. *Manual federation*: the user query is rewritten by the proxy with a SPARQL SERVICE group that tells the query engine which data to obtain from where;
3. *Data offloading*: relevant triples from the original KG are cached in the disposable KG to minimise the need for federation.

The complexity of a user query, the size of a KG and the implementation of the federation algorithm in a given query engine all influence the performance of this step. As could be reasonably expected, in our preliminary experiments with Comunica as query engine, we found that approach 2 consistently outperforms approach 1 for varying user queries and KGs. However, even with manual federation, the amount of probing queries (i.e. COUNT queries) spawned by the query engine can be overwhelming. In our running example, we can either send a user query to the OSM KG with a SERVICE group that pulls computed data from the disposable KG or vice-versa, send the query to the disposable KG and use a SERVICE group to fetch all non-computed data from the OSM KG. Note that not every (public) KG supports federation or has it disabled such that the latter strategy becomes the only option. As the required probing of a KG bears a direct relation with the number of patterns within a SERVICE group, we aim to minimize these patterns. This is achieved via approach 3, where we translate (parts of) basic graph patterns into CONSTRUCT queries that fetch the data required to evaluate them and group it along with triples computed from the virtual predicates in a disposable KG. For instance, relevant triples such as `{ ?mcdo osmt:brand "McDonald's" }` can be offloaded from the OSM KG to the disposable KG. Since offloading eliminates probing queries, it consistently outperformed the automatic and manual federation approaches in our expirements. Of course, this approach is limited by the amount of data to be offloaded which in turn depends on the user query. The decision of when to federate and when to offload is taken by an evolving algorithm that is beyond the scope of this paper.

4. Conclusions

We illustrated how virtual predicates can be used to query a KG along with dynamically computed data using standard SPARQL. Since virtual predicates are indistinguishable from regular predicates, there is no learning curve for SPARQL users. However, this transparency may not be wished for if the computational cost of a virtual predicate is high, hence affecting query performance. To this end, we plan to model the cost estimate as part of an ontology. A direction for future work also includes studying user-based access control to virtual predicates.

5. Demo

For additional information, we refer to our public SPARQL-OTFC Github repository[1]. Here, a demo video, source code and extra documentation can be found.

6. Acknowledgements

References

[1] GraphDB Magic Predicates:
   https://graphdb.ontotext.com/documentation/10.0/pdf/GraphDB.pdf, pp. 696
[2] Jena Property Functions:
   https://jena.apache.org/documentation/query/writing_propfuncs.html
[3] Regalia, Blake et al.: VOLT: A Provenance-Producing, Transparent SPARQL Proxy for the On-Demand Computation of Linked Data and its Application to Spatiotemporally Dependent Data. In: ESWC, pp. 523–538 (2016)
[4] OSM Sophox Service: https://sophox.org
[5] Taelman, Ruben et al.: Comunica: a Modular SPARQL Query Engine for the Web. In: ISWC, pp. 239–255 (2018)
[6] Xiao, Guohui et al.: The Virtual Knowledge Graph System Ontop. In: ISWC, pp. 259—277 (2020)

---

[1]https://github.com/Flanders-Make-vzw/sparql-otfc