Continuously Self-Updating Query Results over Dynamic Heterogeneous Linked Data

Ruben Taelman(✉)

Data Science Lab (Ghent University - iMinds),
Sint-Pietersnieuwstraat 41, 9000 Ghent, Belgium
ruben.taelman@ugent.be

Abstract. Our society is evolving towards massive data consumption from heterogeneous sources, which includes rapidly changing data like public transit delay information. Many applications that depend on dynamic data consumption require highly available server interfaces. Existing interfaces involve substantial costs to publish rapidly changing data with high availability, and are therefore only possible for organisations that can afford such an expensive infrastructure. In my doctoral research, I investigate how to publish and consume real-time and historical Linked Data on a large scale. To reduce server-side costs for making dynamic data publication affordable, I will examine different possibilities to divide query evaluation between servers and clients. This paper discusses the methods I aim to follow together with preliminary results and the steps required to use this solution. An initial prototype achieves significantly lower server processing cost per query, while maintaining reasonable query execution times and client costs. Given these promising results, I feel confident this research direction is a viable solution for offering low-cost dynamic Linked Data interfaces as opposed to the existing high-cost solutions.

Keywords: Linked Data · Triple Pattern Fragments · SPARQL · Continuous querying · Real-time querying

1 Introduction

The Web is an important driver of the increase in data. This data is partially made up of *dynamic* data, which does not remain the same over time, like for example the delay of a certain train or the currently playing track on a radio-station. Dynamic data is mostly published as *data streams* [3], which tend to be offered in a push-based manner. This requires data providers to have a persistent connection with all clients who consume these streams. On top of that, queries over real-time data are expected to be *continuous*, because the data are now continuously updating streams instead of just finite stored datasets. At the same time, this dynamic data also leads to the generation of *historical* data, which may be useful for data analysis.

R. Taelman—Supervised by Ruben Verborgh and Erik Mannens.

H. Sack et al. (Eds.): ESWC 2016, LNCS 9678, pp. 863–872, 2016.
DOI: 10.1007/978-3-319-34129-3_55

In this work, I investigate how to publish and consume non-high frequency real-time and historical Linked Data. This real-time data for example includes sensor results which update at a frequency in the order of seconds, use cases that require updates in the order of milliseconds are excluded. The focus lies at low-cost publication, so that large scale consumption of this data becomes possible without endpoint availability issues.

In the next section, the existing work in the area will be discussed. Section 3 will explain the problem I am trying to solve, after which Sect. 4 will briefly explain the methodology for solving this problem. Section 5 will discuss the evaluation of this solution after which Sect. 6 will present some preliminary results. Finally, in Sect. 7 I will explain the desired impact of this research.

2 State of the Art

Current solutions for querying and publishing dynamic data is divided in the two generally disjunct domains of *stream reasoning* and *versioning*, which will be explained hereafter. After that, a low cost server interface for static data will be explained.

Stream reasoning is defined as "the logical reasoning in real time on gigantic and inevitably noisy data streams in order to support the decision process of extremely large numbers of concurrent users" [4]. This area of research integrates data streams with traditional RDF reasoners. Existing SPARQL extensions for stream processing solutions like C-SPARQL [5] and CQELS [10] are based on *query registration* [4,7], which allows clients to register their query at a streaming-enabled SPARQL endpoint that will continuously evaluate this query. These data streams consist of triples that are *annotated* with a timestamp, which indicates the moment on which the triple is valid. These querying techniques can for example be used to query semantic sensor data [13]. C-SPARQL is a first approach to querying over both static and dynamic data. This solution requires the client to register a query in an extended SPARQL syntax which allows the use of *windows* over dynamic data. The execution of queries is based on the combination of a traditional SPARQL engine with a *Data Stream Management System* (DSMS) [2]. The internal model of C-SPARQL creates queries that distribute work between the DSMS and the SPARQL engine to respectively process the dynamic and static data. CQELS is a "white box" approach, as opposed to the "black box" approaches like C-SPARQL. This means that CQELS natively implements all query operators, as opposed to C-SPARQL that has to transform the query to another language for delegation to its subsystems. This native implementation removes the overhead that black box approaches like C-SPARQL have. The syntax is very similar as to that of C-SPARQL, also supporting query registration and time windows. According to previous research [10], this approach performs much better than C-SPARQL for large datasets, for simple queries and small datasets the opposite is true.

Offering historical data can be achieved by versioning entire datasets [15] using the Memento protocol [14] which extends HTTP with content negotiation in the datetime dimension. Memento adds a new link to resources in the

HTTP header, named the *TimeGate*, which acts as the datetime dimension for a resource. It provides a list of timely versions of the resource which can be requested. Using Memento's datetime content negotiation and TimeGates, it is possible to do *Time Travel* over the web and browse pages at a specific point in time. R&WBase [17] is a triple-store versioning approach based on delta storage combined with traditional snapshots. It offers a method for querying these versioned datasets using SPARQL. The dataset can be retrieved as a virtual graph for each delta revision, thus providing Memento-like time travel without an explicity time indication. TailR [11] provides a platform through which datasets can be versioned based on a combination of snapshot and delta storage and offered using the Memento protocol. It allows queries to retrieve the dataset version at a given time and the times at which a dataset has changed.

Triple Pattern Fragments (TPFS) [18] is a Linked Data publication interface which aims to solve the issue of low availability and performance of existing SPARQL endpoints for static querying. It does this by moving part of the query processing to the client, which reduces the server load at the cost of increased data transfer and potentially increased query evaluation time. The endpoints are limited to an interface with which only separate triple patterns can be queried instead of full SPARQL queries. The client is then responsible for carrying out the remaining work.

3 Problem Statement

Traditional public static SPARQL query endpoints have a major availability issue. Experiments have shown that more than half of them only reach an availability of less than 95 % [6]. The unrestricted complexity of SPARQL queries [12] combined with the public character of SPARQL endpoints requires an enormous server cost, which can lead to a low server availability. Dynamic SPARQL streaming solutions like C-SPARQL and CQELS offer combined access to dynamic data streams and static background data through continuously executing queries. Because of this continuous querying, the cost of these servers can become *even bigger* than with static querying for similarly sized datasets.

The definition of stream reasoning [4] states that it requires reasoning on data streams for "an extremely large number of concurrent users". If we can not even reach a large number of concurrent static SPARQL queries against endpoints without overloading them, how can we expect to do this for dynamic SPARQL queries? Because evaluating these queries put an even greater load on the server if we assume that the continuous execution of a query requires more processing than the equivalent single execution of that query.

The main research question of our work is:

Question 1: How can we combine the low cost publication of non-high frequency real-time and historical data, such that it can efficiently be queried together with static data?

To answer this question, we also need to find an answer to the following questions:

Question 2: How can we efficiently store non-high frequency real-time and historical data and allow efficient transfer to clients?
Question 3: What kind of server interface do we need to enable client-side query evaluation over both static and dynamic data?

These research questions have lead to the following hypotheses:
Hypothesis 1: Our storage solution can store new data in linear time with respect to the amount of new data.
Hypothesis 2: Our storage solution can retrieve data by time or triple values in linear time with respect to the amount of retrieved data.
Hypothesis 3: The server cost for our solution is lower than the alternatives.
Hypothesis 4: Data transfer is the main factor influencing query execution time in relation to other factors like client processing and server processing.

4 Research Approach

As discussed in Sect. 2, TPF is a Linked Data publication interface which aims to solve the high server cost of static Linked Data querying. This is done by partially evaluating queries client-side, which requires the client to break down queries into more elementary queries which can be solved by the limited and low cost TPF server interface. These elementary query results are then locally combined by the client to produce results for the original query.

We will extend this approach to *continuously updating* querying over *dynamic* data.

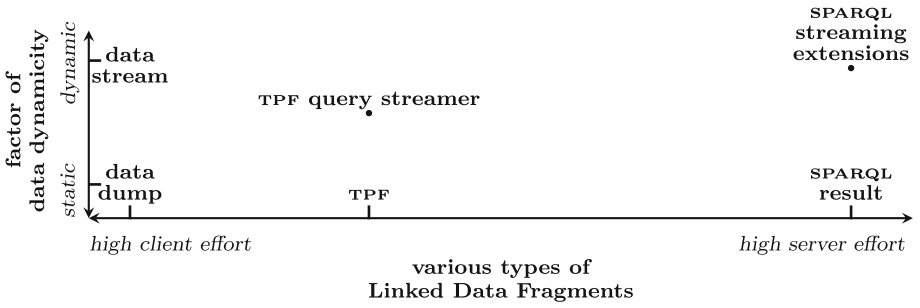

Fig. 1. LDF axis showing the server effort needed to publish different types of interface together with a vertical axis showing the factor of data dynamicity an interface exposes.

Figure 1 shows this shift to more static data in relation to the *Linked Data Fragments (*ldf*)* [19] axis. LDF is a conceptual framework to compare Linked Data publication interface in which TPF can be seen as a trade-off between high server and client effort for data retrieval. SPARQL streaming solutions like C-SPARQL and CQELS can handle high frequency data and they require a high server effort because they are at least as expressive as regular SPARQL. Data streams on the other hand expose high frequency data as well, but here it is the client that has to do most of the work when selecting data from those streams. Our TPF

query streaming extension focuses on non-high frequency data and aims to lower the server effort for more efficient scaling to large numbers of concurrent query executions.

We can split up this research in three parts, which are shown in Fig. 2. First, the server needs to be able to efficiently store dynamic data and publicly offer it. Second, this data must be transferable to the client. Third, a query engine at the client must be able to evaluate queries using this data and keep its answers up to date.

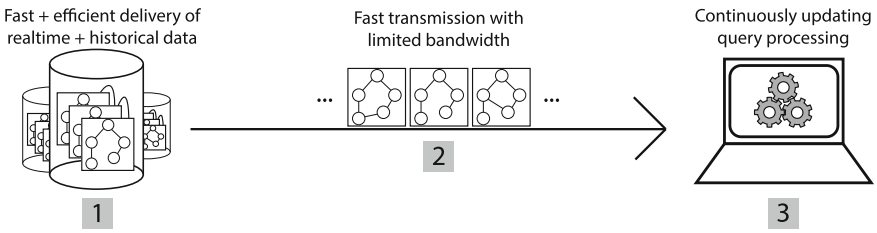

Fig. 2. A client must be able to evaluate queries by retrieving data from multiple heterogeneous datasources.

The storage of historical and real-time data requires a delicate balance between storage size and lookup speed. I will develop a method for this storage and lookup with a focus on the efficient retrieval and storage of versions, the dynamic properties of temporal data and the scalability for historical data. This storage method can be based on the differential storage concept TailR uses, combined with HDT [8] compression for snapshots. The interface through which data will be retrieved could benefit a variant of Memento's *timegate* index to allow users to evaluate historical queries.

For enabling the client to evaluate queries, the client needs to be able to access data from one or more data providers. I will develop a mechanism that enables efficient transmission of temporal data between server and client. By exploiting the similarities between and within temporal versions, I will limit the required bandwidth as much as possible.

To reduce the server cost, the client needs to help evaluating the query. Because of this, we assume that our solution will have a higher client processing cost than streaming-based SPARQL approaches for equivalent queries. For this last goal, I will develop a client-side query engine that is able to do federated querying of temporal data combined with static data against heterogeneous datasources. The engine must keep the real-time results of the query up to date. We can distinguish three requirements for this solution:

– Allowing queries to be declared using a variant of the SPARQL language so that it becomes possible for clients to declare queries over dynamic data. This language should support the RDF stream query semantics that are

being discussed within the RSP Community Group[1]. We could either use the C-SPARQL or CQELS query language, or make a variant if required.
- Building a client-side query engine to do continuous query evaluation, which means that the query results are updated when data changes occur.
- Providing a format for the delivery of continuously updating results for registered queries that will allow applications to handle this data in a dynamic context.

5 Evaluation Plan

I will evaluate each of the three major elements of this research independently: the storage solution for dynamic data at the server; the retrieval of this data and its transmission; and the query evaluation by the client.

5.1 Storage

The evaluation of our storage solution can be done with the help of two experiments.

First, I will execute a large number of insertions of dynamic data against a server. I will measure its CPU usage and determine if it is still able to achieve a decent quality of service for data retrieval. I will also measure the increase in data storage. By analyzing the variance of the CPU usage with different insertion patterns we should be able to accept or reject Hypothesis 1, which states that data can be added in linear time.

The second experiment will consist of the evaluation of data retrieval. This experiment will consist of a large number of lookups against a server by both triple contents and time instants. Doing a variance analysis on the lookup times over these different lookup types will help us to determine the validity of Hypothesis 2, which states that data can be retrieved in linear time.

These two experiments can be combined to see if one or the other demands too much of the server's processing power.

5.2 Retrieval and Transmission

To determine the retrieval cost of data from a server and its transmission, we need to measure the effects of sending a large amount of lookup requests. One of the experiments I performed on the solution that was built during my master's thesis [16] was made up of one server and ten physical clients. Each of these clients could execute from one to ten concurrent unique queries. This results in a series of 10 to 200 concurrent query executions. This setup was used to test the client and server performance of my implementation compared to C-SPARQL and CQELS.

Even though this experiment produced some interesting results, as will be explained in the next section, 200 concurrent clients are not very representative

[1] https://www.w3.org/community/rsp/.

for large scale querying on the public Web. But it can already be used to partially answer Hypothesis 3 that states that our solution has a lower server cost than the alternatives. To extend this, I will develop a mechanism to simulate thousands of simultaneous requests to a server that offers dynamic data. The main bottleneck in the current experiment setup are the query engines on each client. If we were to detach the query engines from the experiment, we could send much more requests to the server and this would result in more representative results. This could be done by first collecting a representative set of HTTP requests that these query engines send to the server. This set of requests should be based on real non-high frequency use cases where it makes sense to have a large number of concurrent query evaluations. These requests can be inspired by existing RSP benchmarks like SRBench [20] and CityBench [1]. Once this collection has been built, the client-CPU intensive task is over, and we can use this pool of requests to quickly simulate HTTP requests to our server. By doing a variance analysis of the server CPU usage for my solution compared to the alternatives, we will be able to determine the truth of Hypothesis 3.

5.3 Query Evaluation

The evaluation of the client side query engine can be done like the experiment of my master's thesis, as explained in the previous section. In this case, the results would be representative since the query engine is expected to be the most resource intensive element in this solution. The CityBench [1] RSP benchmark could for example be used to do measurements based on datasets from various city sensors. By doing a variance analysis of the different client's CPU usage for my solution compared to the alternatives, we will be able to determine how much higher our client CPU cost is than the alternatives. The alternatives in this case include server-side RSP engines like C-SPARQL and CQELS, but also fully client-side stream processing solutions using stream publication techniques like Ztreamy [9]. This way, we test compare our solution with both sides of the LDF axis, on the one hand we have the cases where the server does all of the work while evaluating queries, while on the other hand we have cases where the client does all of the work. For Hypothesis 4, which assumes that data transfer is the main factor for query execution time, we will do a correlation test of bandwidth usage and the corresponding query's execution times.

6 Preliminary Results

During my master's thesis, I did some preliminary research on the topic of continuous querying over non-high frequency real-time data. My solution consisted of *annotating* triples with *time* to give them a timely context, which allowed this dynamic data to be stored on a regular *static* TPF server. An extra layer on top of the TPF client was able to interpret these time-annotated triples as dynamic versions of certain facts. This extra software layer could then derive the exact moment at which the query should be *re-evaluated* to keep its results up to date.

The main experiment that was performed in my master's thesis resulted in the output from Fig. 3. We can see that our approach significantly reduced the server load when compared to C-SPARQL and CQELS, as was the main the goal. The client now pays for the largest part of the query executions, which is caused by the use of TPF. The client CPU usage for our implementation spikes at the time of query initialization because of the rewriting phase, but after that it drops to around 5 %.

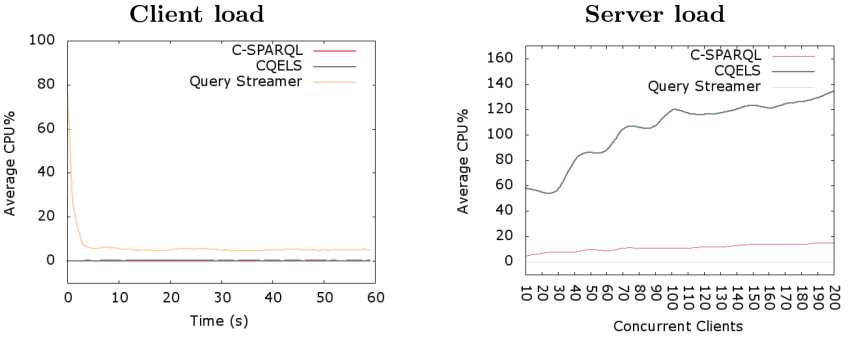

Fig. 3. The client and server CPU usages for one query stream for C-SPARQL, CQELS and our preliminary solution. Our solution has a very low server cost and a higher average client CPU usage when compared to the alternatives.

7 Conclusions

Once we can publish both non-high frequency real-time and historical data at a low server cost, we can finally allow many simultaneous clients to query this data while keeping their results up to date, so this dynamic data can be used in our applications with the same ease as we already do today with static data.

The Semantic Sensor Web already promotes the integration of sensors in the Semantic Web. My solution would make medium to low frequency sensor data queryable on a web-scale, instead of just for a few machines in a private environment for keeping the server cost maintainable.

Current Big Data analysis techniques are able to process data streams, but combining them with other data by discovering semantic relations still remains difficult. The solution presented in this work could make these Big Data analyses possible using Semantic Web techniques. This would make it possible to perform these analyses in a federated manner over heterogeneous sources, since a strength of Semantic Web technologies is the ability to integrate data from the whole web. These analyses could be executed by not only one entity, but all clients with access to the data, while still putting a reasonable load on the server.

References

1. Ali, M.I., Gao, F., Mileo, A.: CityBench: a configurable benchmark to evaluate RSP engines using smart city datasets. In: Arenas, M., et al. (eds.) ISWC 2015. LNCS, vol. 9367, pp. 374–389. Springer, Heidelberg (2015). doi:10.1007/978-3-319-25010-6_25
2. Arasu, A., Babcock, B., Babu, S., Cieslewicz, J., Datar, M., Ito, K., Motwani, R., Srivastava, U., Widom, J.: STREAM: the Stanford data stream management system. Book chapter (2004). http://ilpubs.stanford.edu:8090/641/1/2004-20.pdf
3. Babu, S., Widom, J.: Continuous queries over data streams. ACM Sigmod Rec. **30**(3), 109–120 (2001). http://dl.acm.org/citation.cfm?id=603884
4. Barbieri, D., Braga, D., Ceri, S., Della Valle, E., Grossniklaus, M.: Stream reasoning: where we got so far. In: Proceedings of the NeFoRS2010 Workshop, Co-located with ESWC 2010 (2010). http://wasp.cs.vu.nl/larkc/nefors10/paper/nefors10_paper_0.pdf
5. Barbieri, D.F., Braga, D., Ceri, S., Valle, E.D., Grossniklaus, M.: Querying RDF streams with C-SPARQL. SIGMOD Rec. **39**(1), 20–26 (2010)
6. Buil-Aranda, C., Hogan, A., Umbrich, J., Vandenbussche, P.-Y.: SPARQL web-querying infrastructure: ready for action? In: Alani, H., et al. (eds.) ISWC 2013, Part II. LNCS, vol. 8219, pp. 277–293. Springer, Heidelberg (2013)
7. Della Valle, E., Ceri, S., van Harmelen, F., Fensel, D.: It's a streaming world! Reasoning upon rapidly changing information. IEEE Intell. Syst. **24**(6), 83–89 (2009). http://www.few.vu.nl/ frankh/postscript/IEEE-IS09.pdf
8. Fernández, J.D., Martínez-Prieto, M.A., Gutiérrez, C., Polleres, A., Arias, M.: Binary RDF representation for publication and exchange (HDT). Web Semant. Sci. Serv. Agents World Wide Web **19**, 22–41 (2013). http://www.sciencedirect.com/science/article/pii/S1570826813000036
9. Fisteus, J.A., Garcia, N.F., Fernandez, L.S., Fuentes-Lorenzo, D.: Ztreamy: a middleware for publishing semantic streams on the web. Web Semant. Sci. Serv. Agents World Wide Web **25**, 16–23 (2014)
10. Le-Phuoc, D., Dao-Tran, M., Xavier Parreira, J., Hauswirth, M.: A native and adaptive approach for unified processing of linked streams and linked data. In: Aroyo, L., et al. (eds.) ISWC 2011, Part I. LNCS, vol. 7031, pp. 370–388. Springer, Heidelberg (2011)
11. Meinhardt, P., Knuth, M., Sack, H.: TailR: a platform for preserving history on the web of data. In: Proceedings of the 11th International Conference on Semantic Systems, pp. 57–64. ACM (2015). http://dl.acm.org/citation.cfm?id=2814875
12. Pérez, J., Arenas, M., Gutierrez, C.: Semantics and complexity of SPARQL. In: Cruz, I., Decker, S., Allemang, D., Preist, C., Schwabe, D., Mika, P., Uschold, M., Aroyo, L.M. (eds.) ISWC 2006. LNCS, vol. 4273, pp. 30–43. Springer, Heidelberg (2006)
13. Sheth, A., Henson, C., Sahoo, S.: Semantic sensor web. IEEE Internet Comput. **12**(4), 78–83 (2008). http://corescholar.libraries.wright.edu/cgi/viewcontent.cgi?article=2125&context=knoesis
14. de Sompel, H.V., Nelson, M.L., Sanderson, R., Balakireva, L., Ainsworth, S., Shankar, H.: Memento: time travel for the web. CoRR abs/0911.1112 (2009). http://arxiv.org/abs/0911.1112
15. de Sompel, H.V., Sanderson, R., Nelson, M.L., Balakireva, L., Shankar, H., Ainsworth, S.: An HTTP-based versioning mechanism for linked data. CoRR abs/1003.3661 (2010). http://arxiv.org/abs/1003.3661

16. Taelman, R.: Continuously updating queries over real-time linked data. Master's thesis, Ghent University, Belgium (2015). http://rubensworks.net/raw/publications/2015/continuously_updating_queries_over_real-time_linked_data.pdf
17. Vander Sande, M., Colpaert, P., Verborgh, R., Coppens, S., Mannens, E., Van de Walle, R.: R&Wbase: git for triples. In: LDOW (2013). http://events.linkeddata.org/ldow2013/papers/ldow2013-paper-01.pdf
18. Verborgh, R., Hartig, O., De Meester, B., Haesendonck, G., De Vocht, L., Vander Sande, M., Cyganiak, R., Colpaert, P., Mannens, E., Van de Walle, R.: Querying datasets on the Web with high availability. In: Proceedings of the 13th International Semantic Web Conference (2014). http://linkeddatafragments.org/publications/iswc2014.pdf
19. Verborgh, R., Vander Sande, M., Colpaert, P., Coppens, S., Mannens, E., Van de Walle, R.: Web-scale querying through Linked Data Fragments. In: Proceedings of the 7th Workshop on Linked Data on the Web (2014). http://events.linkeddata.org/ldow2014/papers/ldow2014_paper_04.pdf
20. Zhang, Y., Duc, P.M., Corcho, O., Calbimonte, J.-P.: SRBench: a streaming RDF/SPARQL benchmark. In: Heflin, J., et al. (eds.) ISWC 2012, Part I. LNCS, vol. 7649, pp. 641–657. Springer, Heidelberg (2012)