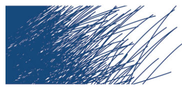

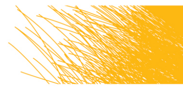

biblio.ugent.be

The UGent Institutional Repository is the electronic archiving and dissemination platform for all UGent research publications. Ghent University has implemented a mandate stipulating that all academic publications of UGent researchers should be deposited and archived in this repository. Except for items where current copyright restrictions apply, these papers are available in Open Access.

This item is the archived peer-reviewed author-version of:

Moving Real-Time Linked Data Query Evaluation to the Client

Ruben Taelman, Ruben Verborgh, Pieter Colpaert, Erik Mannens, and Rik Van de Walle

In: Proceedings of the 13th Extended Semantic Web Conference: Posters and Demos, 2016.

To refer to or to cite this work, please use the citation to the published version:

Taelman, R., Verborgh, R., Colpaert, P., Mannens, E., and Van de Walle, R. (2016). Moving Real-Time Linked Data Query Evaluation to the Client. ***Proceedings of the 13th Extended Semantic Web Conference: Posters and Demos***

Moving Real-Time Linked Data Query Evaluation to the Client

Ruben Taelman, Ruben Verborgh, Pieter Colpaert,
Erik Mannens, and Rik Van de Walle

Data Science Lab (Ghent University - iMinds)
Sint-Pietersnieuwstraat 41, 9000 Ghent, Belgium
{firstname.lastname}@ugent.be

Abstract. Traditional RDF stream processing engines work completely server-side, which contributes to a high server cost. For allowing a large number of concurrent clients to do continuous querying, we extend the low-cost Triple Pattern Fragments (TPF) interface with support for time-sensitive queries. In this poster, we give the overview of a client-side RDF stream processing engine on top of TPF. Our experiments show that our solution significantly lowers the server load while increasing the load on the clients. Preliminary results indicate that our solution moves the complexity of continuously evaluating real-time queries from the server to the client, which makes real-time querying much more scalable for a large amount of concurrent clients when compared to the alternatives.

Keywords: Linked Data, Linked Data Fragments, SPARQL, continuous querying, real-time querying

1 Introduction

Several use cases need updating query results over time, and may thus require the re-execution of entire queries over and over again (i.e., polling). The problem is that polling can be very inefficient when not knowing when the data will change. An additional problem is that many public (even static) SPARQL query endpoints suffer from low availability [2]. This is partially caused by the unrestricted complexity of SPARQL queries [5] combined with the public character of SPARQL endpoints. RDF stream processing engines like C-SPARQL [1] and CQELS [4] offer combined access to dynamic data streams and static background data through continuously executing queries. Because of this continuous querying, the cost for these servers is *even higher* than with static querying.

In this work we present a client-side RDF stream processing engine based on Triple Pattern Fragments (TPF) [6]. TPF is a low-cost server interface for retrieving triple patterns, this makes it possibly for a client to evaluate any query by breaking it up into several triple patterns and joining them locally. We focus on non-high-frequency dynamic data, for example, information on train delays, which updates in the order of minutes. Because some dynamic data might have a frequency that is too high for clients to efficiently poll. The resulting framework requires the server to *annotate* its data with a predicted expiration time. Using this expiration time, the client can efficiently determine when

to retrieve fresh data. The generic approach in this paper is applied to the use case of public transit route planning. It can be used in various other domains with continuously updating data, such as smart city dashboards, business intelligence, or sensor networks.

2 Query Streamer

Our solution consists of a partial redistribution of query evaluation workload from the server to the client. This requires the client to be able to access the server data so that the query evaluation can be done client-side. There needs to be a distinction between regular static data and continuously updating dynamic data in the server's dataset. By *annotating* dynamic data with a time interval or expiration time using a *temporal vocabulary* [3], the client can detect for how long a certain fact remains valid. The data could however still remain the same after its expiration. When dynamic data expires in time, the client knows that it has to evaluate the query again to fetch the latest version of the data.

We have added an extra layer, which is called the *Query Streamer*, on top of the TPF client. This query streamer is able to transform a regular SPARQL query to a separate *static* and *dynamic* query. This rewriting is done by exchanging metadata with the server. The Query Streamer continuously evaluates this dynamic query based on the time annotation it can find on the dynamic data. The Query Streamer exploits these time annotations by only initiating new queries when the dynamic facts have expired. Every time results from the dynamic query are finalized, they are combined with the static query to form a *materialized static query*. This materialized static query will then either be evaluated or its results will be retrieved from a local cache. The combined results of the dynamic query and materialized static query are then continuously returned to the user who initiated the original query.

3 Preliminary Evaluation

In order to analyze the effects of our solution, we set up an experiment to measure the impact of our proposed redistribution of workload between the client and server by simultaneously executing a set of queries against a TPF server using our proposed solution. We repeat this experiment for two state-of-the-art server-side solutions: C-SPARQL and CQELS, in which the clients simply register the query to the respective server engine and get a stream of results.

To test the client and server performance, our experiment consists of one server and ten physical clients. Each of these clients can execute from one to twenty unique concurrent queries derived from the query in Listing 1.1. This results in a series of 10 to 200 concurrent query executions.

Our solution was implemented[1] in JavaScript using Node.js to allow for easy communication with the existing TPF client. The tests[2] were executed on machines having

[1] The source code for this implementation is available at `https://github.com/rubensworks/TPFStreamingQueryExecutor`

[2] The code used to run these experiments with the relevant queries can be found at `https://github.com/rubensworks/TPFStreamingQueryExecutor-experiments/`

```
SELECT ?delay ?platform ?headSign ?routeLabel ?departureTime
WHERE {
    _:id t:delay         ?delay.
    _:id t:platform      ?platform.
    _:id t:departureTime ?departureTime.
    _:id t:headSign      ?headSign.
    _:id t:routeLabel    ?routeLabel.
    FILTER (?departureTime > "2015-12-08T10:20:00"^^xsd:dateTime).
    FILTER (?departureTime < "2015-12-08T11:20:00"^^xsd:dateTime).
}
```

Listing 1.1: The basic SPARQL query for retrieving all upcoming train departure information in a certain station. The two first triple patterns are dynamic, the last three are static.

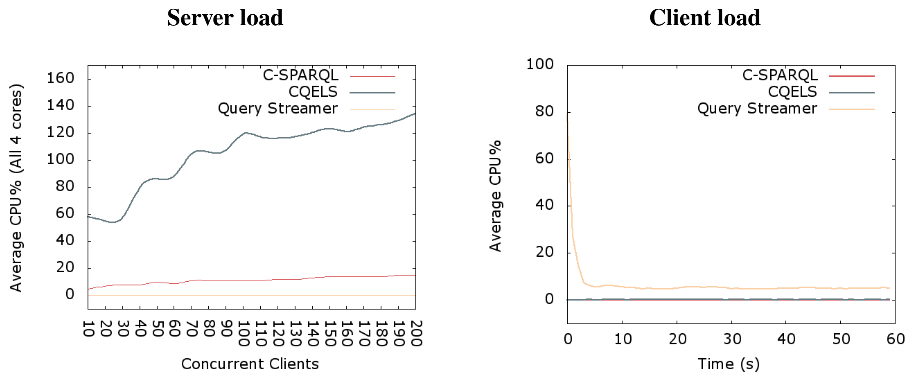

Fig. a: The server CPU usage of our solution proves to be influenced less by the number of clients.

Fig. b: In the case of 200 concurrent clients, client CPU usage initially is high after which it converges to about 5%. The usage for C-SPARQL and CQELS is almost non-existing.

Fig. 1: Average server and client CPU usage for one query stream for C-SPARQL, CQELS and the proposed solution. Our solution effectively moves complexity from the server to the client.

two Hexacore Intel E5645 (2.4GHz) CPUs with 24 GB RAM and were running Ubuntu 12.04 LTS.

The server performance results from our main experiment can be found in Figure 1a. This plot shows an increasing CPU usage for C-SPARQL and CQELS for higher numbers of concurrent query executions. On the other hand, our solution never reaches more than 1 percent of server CPU usage.

The results for the average CPU usage across the duration of the query evaluation of all clients that send queries to the server in our main experiment can be found in Figure 1b. The clients that send C-SPARQL and CQELS queries to the server have a client CPU usage of nearly zero percent for the whole duration of the query evaluation. The clients using the client-side query streamer solution that is presented in this work have an initial CPU peak reaching about 80%, which drops to about 5% after 4 seconds. This initial peak is caused by the preprocessing done by our query streamer.

4 Conclusions

In this paper, we presented a solution for doing client-side query evaluation over dynamic data, with the goal of lowering the server load. Our preliminary evaluation shows that for queries of limited complexity with limited dataset sizes, our solution significantly reduces the server load. This makes it possible for the server to handle much more client requests when compared to alternative approaches. This lower server load consequently leads to a higher client load in our experiments. Future research should show how this solution performs for larger datasets and different query types. The movement from query registration at the server as is done by C-SPARQL and CQELS to client-side query evaluation is important for reducing the server load and for being able to publish dynamic data at a low cost.

This low-cost publication of dynamic Linked Data opens up a whole new range of possibilities. Dynamic data that currently requires expensive server infrastructure for its publication to a large number of potential clients or is somehow being rate-limited to avoid server overloading, can now be exposed through a low-cost interface where clients are required to do part of the work. This can be used, for example, for public access to real-time information on public transport scheduling and non-high frequency sensors.

Acknowledgments The described research activities were funded by iMinds and Ghent University, the Institute for the Promotion of Innovation by Science and Technology in Flanders (IWT), the Fund for Scientific Research Flanders (FWO Flanders), and the European Union. Ruben Verborgh is a Postdoctoral Fellow of the Research Foundation Flanders.

References

1. Barbieri, D.F., Braga, D., Ceri, S., Valle, E.D., Grossniklaus, M.: Querying RDF streams with C-SPARQL. SIGMOD Rec. 39(1), 20–26 (Sep 2010)
2. Buil-Aranda, C., Hogan, A., Umbrich, J., Vandenbussche, P.Y.: SPARQL web-querying infrastructure: Ready for action? In: The Semantic Web–ISWC 2013, pp. 277–293 (2013)
3. Gutierrez, C., Hurtado, C., Vaisman, A.: Introducing time into RDF. Knowledge and Data Engineering, IEEE Transactions on 19(2), 207–218 (Feb 2007)
4. Le-Phuoc, D., Dao-Tran, M., Parreira, J.X., Hauswirth, M.: A native and adaptive approach for unified processing of linked streams and Linked Data. In: The Semantic Web–ISWC 2011, pp. 370–388 (2011)
5. Pérez, J., Arenas, M., Gutierrez, C.: Semantics and complexity of SPARQL. In: International semantic web conference. vol. 4273, pp. 30–43 (2006)
6. 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)