Continuous Client-side Query Evaluation over Dynamic Linked Data

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. Existing solutions to query dynamic Linked Data sources extend the SPARQL language, and require continuous server processing for each query. Traditional SPARQL endpoints already accept highly expressive queries, so extending these endpoints for time-sensitive queries increases the server cost even further. To make continuous querying over dynamic Linked Data more affordable, we extend the low-cost Triple Pattern Fragments (TPF) interface with support for time-sensitive queries. In this paper, we introduce the TPF Query Streamer that allows clients to evaluate SPARQL queries with continuously updating results. Our experiments indicate that this extension significantly lowers the server complexity, at the expense of an increase in the execution time per query. We prove that by moving the complexity of continuously evaluating queries over dynamic Linked Data to the clients and thus increasing bandwidth usage, the cost at the server side is significantly reduced. Our results show that this solution makes real-time querying 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

As the Web of Data is a *dynamic* dataspace, different results may be returned depending on when a question was asked. The end-user might be interested in seeing the query results update over time, for instance, by re-executing the entire query over and over again ("polling"). This is, however, not very practical, especially if it is unknown beforehand when data will change. An additional problem is that many public (even static) SPARQL query endpoints suffer from a low availability [5]. The unrestricted complexity of SPARQL queries [15] combined with the public character of SPARQL endpoints entails a high server cost, which makes it expensive to host such an interface with high availability. *Dynamic* SPARQL streaming solutions 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 therefore devise a solution that enables clients to continuously evaluate non-high frequency queries by polling specific fragments of the data. The resulting framework performs this without the server needing to remember any client state.

Its mechanism requires the server to *annotate* its data so that 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. This paper extends our earlier work [17] with additional experiments.

In the next section, we discuss related research on which our solution will be based. After that, Section 3 gives a general problem statement. In Section 4, we present a motivating use case. Section 5 discusses different techniques to represent dynamic data, after which Section 6 gives an explanation of our proposed query solution. Next, Section 7 shows an overview of our experimental setup and its results. Finally, Section 8 discusses the conclusions of this work with further research opportunities.

2   Related Work

In this section, we first explain techniques to perform RDF annotation, which will be used to determine freshness. Then, we zoom in on possible representations of temporal data in RDF. We finish by discussing existing SPARQL streaming extensions and a low-cost (static) Linked Data publication technique.

2.1   RDF Annotations

Annotations allow us to attach metadata to triples. We might for example want to say that a triple is only valid within a certain time interval, or that a triple is only valid in a certain geographical area.

RDF 1.0 [11] allows triple annotation through *reification*. This mechanism uses *subject*, *predicate*, and *object* as predicates, which allow the addition of annotations to such reified RDF triples. The downside of this approach is that one triple is now transformed to three triples, which significantly increases the total amount of triples.

Singleton Properties [14] create unique instances (singletons) of predicates, which then can be used for further specifying that relationship, for example, by adding annotations. New instances of predicates are created by relating them to the old predicate through the `sp:singletonPropertyOf` predicate. While this approach requires fewer triples than reification to represent the same information, it still has the issue of the original triple being lost, because the predicate is changed in this approach.

With RDF 1.1 [6] came *graph* support, which allows triples to be encapsulated into named graphs, which can also be annotated. Graph-based annotation requires fewer triples than both reification and singleton properties when representing the same information. It requires the addition of a fourth element to the triple which transforms it to a quad. This fourth element, the *graph*, can be used to add the annotations to.

2.2   Temporal data in the RDF model

Regular RDF triples cannot express the time and space in which the fact they describe is true. In domains where data needs to be represented for certain times or time ranges, these traditional representations should thus be extended. There are two main mechanisms

for adding time [9]. *Versioning* will take snapshots of the complete graph every time a change occurs. *Time labeling* will annotate triples with their change time. The latter is believed to be a better approach in the context of RDF, because complete snapshots introduce overhead, especially if only a small part of the graph changes. Gutierrez et al. made a distinction between *point-based* and *interval-based* labeling, which are interchangeable [8]. The former states information about an element at a certain time instant, while the latter states information at all possible times between two time instants.

The same authors introduced a *temporal vocabulary* [8] for the discussed mechanisms, which will be referred to as `tmp` in the remainder of this document. Its core predicates are:

`tmp:interval` This predicate can be used on a subject to make it valid in a certain time interval. The range of this property is a time interval, which is represented by the two mandatory properties `tmp:initial` and `tmp:final`.

`tmp:instant` Used on subjects to make it valid on a certain time instant as a point-based time representation. The range of this property is `xsd:dateTime`.

`tmp:initial` and `tmp:final` The domain of these predicates is a time interval. Their range is a `xsd:dateTime`, and they respectively indicate the start and the end of the interval-based time representation.

Next to these properties, we will also introduce our own predicate `tmp:expiration` with range `xsd:dateTime` which indicates that the subject is only valid up until the given time.

2.3 SPARQL Streaming Extensions

Several SPARQL extensions exist that enable querying over data streams. These data streams are traditionaly represented as a monotonically non-decreasing stream of triples that are annotated with their timestamp. These require *continuous processing* [7] of queries because of the constantly changing data.

C-SPARQL [4] is an approach to querying over static and dynamic data. This system requires the client to *register* a query to the server in an extended SPARQL syntax that allows the use of *windows* over dynamic data. This *query registration* [3,7] must occur by clients to make sure that the streaming-enabled SPARQL endpoint can continuously re-evaluate this query, as opposed to traditional endpoints where the query is evaluated only once. A *window* [2] is a subsection of facts ordered by time so that not all available information has to be taken into account while processing. These windows can have a certain size which indicates the time range and is advanced in time by a *stepsize*. C-SPARQL's execution of queries is based on the combination of a regular 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 [12] is a "white box" approach, as opposed to "black box" approaches like C-SPARQL. This means that CQELS natively implements all query operators without transforming it to another language, removing the overhead of delegating it to another system. The syntax is similar to that of C-SPARQL, also supporting query registration and time windows. According to previous research [12], CQELS performs much better than C-SPARQL for large datasets; for simple queries and small datasets the opposite is true.

2.4   Triple Pattern Fragments

Experiments have shown that more than half of public SPARQL endpoints have an availability of less than 95% [5]. Any number of clients can send arbitrarily complex SPARQL queries, which could form a bottleneck in endpoints. *Triple Pattern Fragments* (TPF) [18] aim to solve this issue of high interface cost by moving part of the query evaluation to the client, which reduces the server load, at the cost of increased query times and bandwidth. The purposely limited interface only accepts separate triple pattern queries. Clients can use it to evaluate more complex SPARQL queries locally, also over federations of interfaces [18].

3   Problem Statement

In order to lower server load during continuous query evaluation, we move a significant part of the query evaluation from server to client. We annotate dynamic data with their valid time to make it possible for clients to derive an optimal query evaluation frequency. For this research, we identified the following research questions:

Question 1: Can clients use volatility knowledge to perform more efficient continuous SPARQL query evaluation by polling for data?

Question 2: How does the client and server load of our solution compare to alternatives?

Question 3: How do different time-annotation methods perform in terms of the resulting execution times?

These research questions lead to the following hypotheses:

Hypothesis 1: The proposed framework has a lower server cost than alternatives.

Hypothesis 2: The proposed framework has a higher client cost than streaming-based SPARQL approaches for equivalent queries.

Hypothesis 3: Client-side caching of static data reduces the execution times proportional to the fraction of static triple patterns that are present in the query.

4   Use Case

A guiding use case, based on public transport, will be referred to in the remainder of this paper. When public transport route planning applications return dynamic data, they can account for factors such as train delays as part of a continuously updating route plan. In this use case, different clients need to obtain all train departure information for a certain station. This requires the following concepts:
  1. **Departure** (*static*): Unique URI for the departure of a certain train.
  2. **Headsign** (*static*): The label of the train showing its destination.
  3. **Departure Time** (*static*): The *scheduled* departure time of the train.
  4. **Route Label** (*static*): The identifier for the train and its route.
  5. **Delay** (*dynamic*): The delay of the train, which can increase through time.

```
@prefix t: <http://example.org/train/>.
@prefix td: <http://example.org/traindata/>.
td:departure-48 t:delay         "0S"^^xsd:xs:duration;
                t:platform      td:platform-1a;
                t:departureTime "2014-12-05T10:37:00+01:00"^^xsd:dateTimeStamp;
                t:headSign      "Ghent";
                t:routeLabel    "IC 1831".
```
Listing 1.1: Train information with static time information according to the basic data model.

```
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 1.2: 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.

6. **Platform** (*dynamic*): The platform number of the station at which the train will depart, which can be changed through time if delays occur.

Listing 1.1 shows example data in this model. The SPARQL query in Listing 1.2 can retrieve all information using this basic data model.

5 Dynamic Data Representation

Our solution consists of a partial redistribution of query evaluation workload from the server to the client, which requires the client to be able to access the server data. There needs to be a distinction between regular static data and continuously updating dynamic data in the server's dataset. For this, we chose to define a certain temporal range in which these dynamic facts are valid, as a consequence the client will know when the data becomes invalid and has to fetch new data to remain up-to-date. To capture the temporal scope of data triples, we annotate this data with time. In this section, we discuss two different types of time labeling, and different methods to annotate this data.

5.1 Time Labeling Types

We use interval-based labeling to indicate the *start and endpoint* of the period during which triples are valid. Point-based labeling is used to indicate the *expiration time*.

With expiration times, we only save the latest version of a given fact in a dataset, assuming that the old version can be removed when a newer one arrives. These expiration times provide enough information to determine when a certain fact becomes invalid in time. We use time intervals for storing multiple versions of the same fact, i.e., for maintaining a history of facts. These time intervals must indicate a start- and endtime for making it possible to distinguish between different versions of a certain fact. These

intervals cannot overlap in time for the same facts. When data is volatile, consecutive interval-based facts will accumulate quickly. Without techniques to aggregate or remove old data, datasets will quickly grow, which can cause increasingly slower query executions. This problem does not exist with expiration times because in this approach we decided to only save the latest version of a fact, so this volatility will not have any effect on the dataset size.

5.2   Methods for Time Annotation

The two time labeling types introduced in the last section can be annotated on triples in different ways. In Section 2.1 we discussed several methods for RDF annotation. We will apply time labels to triples using the singleton properties, graphs and implicit graphs annotation techniques.

Singleton Properties   *Singleton properties* annotation is done by creating a singleton property for the predicate of each dynamic triple. Each of these singleton properties can then be annotated with its time annotation, being either a time interval or expiration times.

Graphs   To time-annotate triples using *graphs*, we can encapsulate triples inside contexts, and annotate each context graph with a time annotation.

Implicit Graphs   A TPF interface gives a unique URI to each fragment corresponding to a triple pattern, including patterns without variables, i.e., actual triples. Since Triple Pattern Fragments [18] are the basis of our solution, we can interpret each fragment as a graph. We will refer to these as *implicit graphs*. This URI can then be used as graph identifier for this triple for adding time information. For example, the URI for the triple `<s> <p> <o>` on the TPF interface located at `http://example.org/dataset/` is `http://example.org/dataset?subject=s&predicate=p&object=o`.

The choice of time annotation method for publishing temporal data will also depend on its capability to *group* time labels. If certain dynamic triples have identical time labels, these annotations can be shared to further reduce the required amount of triples if we are using singleton properies or graphs. When we would have three train delay triples which are valid for the same time interval using graph annotation, these three triples can be placed in the same graph. This will make sure they refer to the same time interval without having to replicate this annotation two times more. In the case of implicit graph annotation, this grouping of triples is not possible, because each triple has a unique graph identifier determined by the interface. This would be possible if these different identifiers are linked to each other with for example `sameAs` relationships that our query engine takes into account, which would introduce further overhead.

We will execute our use case for each of these annotation methods. In practise, an annotation method must be chosen depending on the requirements and available technologies. If we have a datastore that supports quads, graph-based annotation is the best choice because of it requires the least amount of triples. If our datastore does not support quads, we can use singleton properties. If we have a TPF-like interface at which

our data is hosted, we can use implicit graphs as annotation technique, if however many of those triples can be grouped under the same time label, singleton properties are a better alternative because the latter has grouping support.

6 Query Engine

TPF query evaluation involves server and client software, because the client actively takes part in the query evaluation, as opposed to traditional SPARQL endpoints where the server does all of the work. Our solution allows users to send a normal SPARQL query to the local query engine which autonomously detects the dynamic parts of the query and continuously sends back results from that query to the user. In this section, we discuss the architecture of our proposed solution and the most important algorithms that were used to implement this.

6.1 Architecture

Our solution must be able to handle regular SPARQL 1.1 queries, detect the dynamic parts, and produce continuously updating results for non-high frequency queries. To achieve this, we chose to build an extra software layer on top of the existing TPF client that supports each discussed labeling type and annotation method and is capable of doing dynamic query transformation and result streaming. At the TPF server, dynamic data must be annotated with time depending on the used combination of labeling type and method. The server expects dynamic data to be pushed to the platform by an external process with varying data. In the case of graph-based annotation, we have to extend the TPF server implementation, so that it supports quads. This dynamic data should be pushed to the platform by an external process with varying data.

Figure 1 shows an overview of the architecture for this extra layer on top of the TPF client, which will be called the *TPF Query Streamer* from now on. The left-hand side shows the *User* that can send a regular SPARQL query to the TPF Query Streamer entry-point and receives a stream of query results. The system can execute queries through the local *Basic Graph Iterator*, which is part of the TPF client and executes queries against a TPF server.

The TPF Query Streamer consists of six major components. First, there is the *Rewriter* module which is executed only once at the start of the query streaming loop. This module is able to transform the original input query into a *static* and a *dynamic query* which will respectively retrieve the static background data and the time-annotated changing data. This transformation happens by querying metadata of the triple patterns against the entry-point through the local TPF client. The *Streamer* module takes this dynamic query, executes it and forwards its results to the *Time Filter*. The *Time Filter* checks the time annotation for each of the results and rejects those that are not valid for the current time. The minimal expiration time of all these results is then determined and used as a delayed call to the *Streamer* module to continue with the *streaming loop*, which is determined by the repeated invocation of the *Streamer* module. This minimal expiration time will
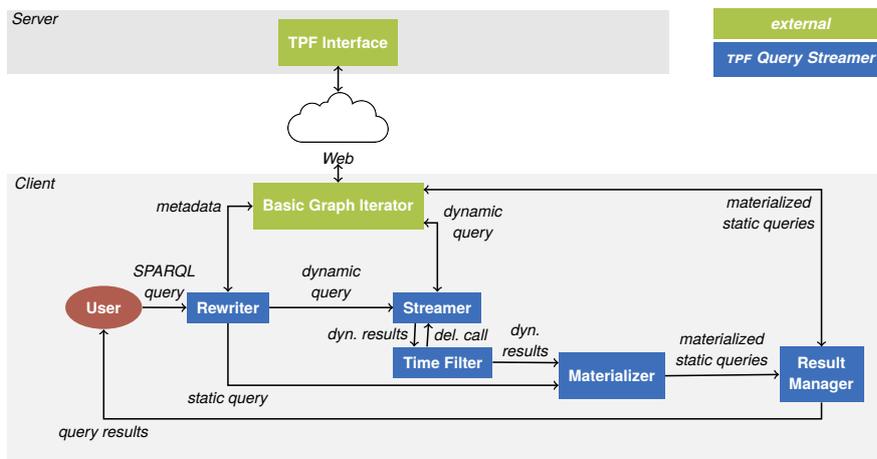
Fig. 1: Overview of the proposed client-server architecture.

make sure that when at least one of the results expire, a new set of results will be fetched as part of the next query iteration. The filtered dynamic results will be passed on to the *Materializer* which is responsible for creating *materialized static queries*. This is a transformation of the *static query* with the dynamic results filled in. These *materialized static queries* are passed to the *Result Manager* which is able to cache these queries. Finally, the *Result Manager* retrieves previous *materialized static query* results from the local cache or executes this query for the first time and stores its results in the cache. These results are then sent to the client who had initiated continuous query.

6.2   Algorithms

Query rewriting As mentioned in the previous section, the *Rewriter* module performs a preprocessing step that can transform a regular SPARQL 1.1 query into a static and dynamic query. A first step in this transformation is to detect which triple patterns inside the original query refer to static triples and which refer to dynamic triples. We detect this by making a separate query for each of the triple patterns and transforming each of them to a dynamic query. An example of such a transformation can be found in Listing 1.3. We then evaluate each of these transformed queries and assume a triple pattern is *dynamic* if its corresponding query has at least one result. Another step before the actual query splitting is the conversion of blank nodes to variables. We will end up with one static query and one dynamic query, in case these graphs were originally connected, they still need to be connected after the query splitting. This connection is only possible with variables that are visible, meaning that these variables need to be part of the SELECT clause. However, a variable can also be anonymous and not visible: these are blank nodes. To make sure that we take into account blank nodes that connect the static and dynamic graph, these nodes have to be converted to variables, while maintaining their semantics. After this step, we iterate over each triple pattern of the original query and assign them to either the static or the dynamic query depending on whether or not the pattern is respectively static or dynamic. This assignment must maintain the hierarchical

```
SELECT ?s ?p ?o ?time WHERE {
    GRAPH ?g0 { ?s ?p ?o }
    ?g0 tmp:expiration ?time
}
```
Listing 1.3: Dynamic SPARQL query for the triple pattern ?s ?p ?o for graph-based annotation with expiration times.

structure of the original query, in some cases this causes triple patterns to be present in the dynamic query when using complex operators like UNION to maintain correct query semantics. An example of this query transformation for our basic query from Listing 1.2 can be found in Listings 1.4 and 1.5.

Query materialization The *Materializer* module is responsible for creating *materialized static queries* from the static query and the current dynamic query results. This is done by filling in each dynamic result into the static query variables. It is possible that multiple results are returned from the dynamic query evaluation, which is the same amount of materialized static queries that can be derived. Assuming that we, for example, find the following single dynamic query result from the dynamic query in Listing 1.5: {?id ↦ <http://example.org/train#train4815>, ?delay ↦ "P10S"ˆˆxsd:duration} then we can derive the materialized static query by filling in these two variables into the static query from Listing 1.4, the resulting query can be found in Listing 1.6.

Caching The *Result manager* is the last step in the streaming loop for returning the materialized static query results of one time instance. This module is responsible for either getting results for given queries from its cache, or fetching the results from the TPF client. First, an identifier will be determined for each materialized static query. This identifier will serve as a key to cache static data and should correctly and uniquely identify static results based on dynamic results. This is equivalent to saying that this identifier should be the *connection* between the static and dynamic graphs. This connection is the intersection of the variables present in the WHERE clause of the static and dynamic queries. Since the dynamic query results are already available at this point, these variables all have values, so this cache identifier can be represented by these variable results. The graph connection between the static and dynamic queries from Listing 1.4 and Listing 1.5 is ?id. The cache identifier for a result where ?id is "train:4815" is for example "?id=train:4815".

```
SELECT ?id ?headSign ?routeLabel ?departureTime
WHERE {
    ?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 1.4: Static SPARQL query which has been derived from the basic SPARQL query from Listing 1.2 by the *Rewriter* module.

```
SELECT ?id ?delay ?platform ?final0 ?final1
WHERE {
    GRAPH ?g0 { ?id t:delay ?delay. }
    ?g0 tmp:expiration ?final0.
    GRAPH ?g1 { ?id t:platform ?platform. }
    ?g1 tmp:expiration ?final1.
}
```

Listing 1.5: Dynamic SPARQL query which has been derived from the basic SPARQL query from Listing 1.2 by the *Rewriter* module. Graph-based annotation is used with expiration times.

```
SELECT ?headSign ?routeLabel ?departureTime
WHERE {
    <http://example.org/train#train4815> t:departureTime ?departureTime.
    <http://example.org/train#train4815> t:headSign       ?headSign.
    <http://example.org/train#train4815> t:routeLabel     ?routeLabel.
    FILTER (?departureTime > "2015-12-08T10:20:00"^^xsd:dateTime).
    FILTER (?departureTime < "2015-12-08T11:20:00"^^xsd:dateTime).
}
```

Listing 1.6: Materialized static SPARQL query derived by filling in the dynamic query results into the static query from Listing 1.6.

7   Evaluation

In order to validate our hypotheses from Section 3, 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 server using our proposed solution. We repeat this experiment for two state-of-the-art solutions: C-SPARQL and CQELS.

To test the client and server performance, our experiment consisted of one server and ten physical clients. Each of these clients can execute from one to twenty unique concurrent queries based on the use case from Section 4. The data for this experiment was derived from real-world Belgian railway data using the iRail API[1]. This results in a series of 10 to 200 concurrent query executions. This setup was used to test the client and server performance of different SPARQL streaming approaches.

For comparing the efficiency of different time annotation methods and for measuring the effectiveness of our client-side cache, we measured the execution times of the query for our use case from Section 4. This measurement was done for different annotation methods, once with the cache and once without the cache. For discovering the evolution of the query evaluation efficiency through time, the measurements were done over each query stream iteration of the query.
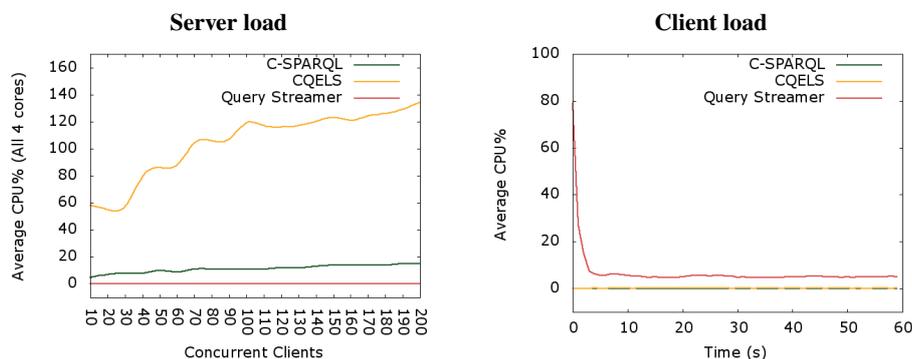
The discussed architecture was implemented[2] in JavaScript using Node.js to allow for easy communication with the existing TPF client.

The tests[3] were executed on the Virtual Wall (generation 2) environment from iMinds [10]. Each machine had two Hexacore Intel E5645 (2.4GHz) CPUs with 24 GB RAM

---

[1] https://hello.irail.be/api/1-0/

[2] The source code for this implementation is available at https://github.com/LinkedDataFragments/QueryStreamer.js/tree/eswc2016

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

Server load

Client load

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.
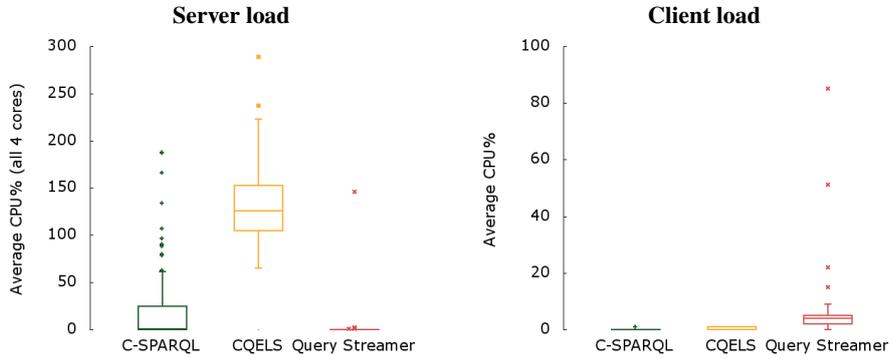
Fig. 2: Average server and client CPU usage for one query stream for C-SPARQL, CQELS and the proposed solution. Our solution effectively moves complexity from the server to the client.

and was running Ubuntu 12.04 LTS. For CQELS, we used version 1.0.1 of the engine [13]. For C-SPARQL, this was version 0.9 [16]. The dataset for this use case consisted of about 300 static triples, and around 200 dynamic triples that were created and removed each ten seconds. Even this relatively small dataset size already reveals important differences in server and client cost, as we will discuss in the paragraphs below.

Server Cost The server performance results from our main experiment can be seen in Figure 2a. 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 one percent of server CPU usage. Figure 3a shows a detailed view on the measurements in the case of 200 simultaneous query executions: the CPU peaks for the alternative approaches are much higher and more frequent than for our solution.

Client Cost The results for the average CPU usage across the duration of the query evaluation of all clients that sent queries to the server in our main experiment can be seen in Figures 2b and 3b. The clients that were sending C-SPARQL and CQELS queries to the server had a client CPU usage of nearly zero percent for the whole duration of the query evaluation. The clients using the client-side TPF Query Streamer solution that was presented in this work had an initial CPU peak reaching about 80%, which dropped to about 5% after 4 seconds.

Annotation Methods The execution times for the different annotation methods, once with and once without cache can be seen in Figure 4. The three annotation methods have about the same relative performance in all figures, but the execution times are generally lower in the case where the client-side cache was used, except for the first query iteration. The execution times for expiration time annotation when no cache is used are constant, while the execution times with caching slightly decrease over time.

Server load

Client load

Fig. a: Server CPU peaks for C-SPARQL and CQELS compared to our solution.

Fig. b: Client CPU usage for our solution is significantly higher.

Fig. 3: Detailed view on all server and client CPU measurements for C-SPARQL, CQELS and the solution presented in this work for 200 simultaneous query evaluations against the server.
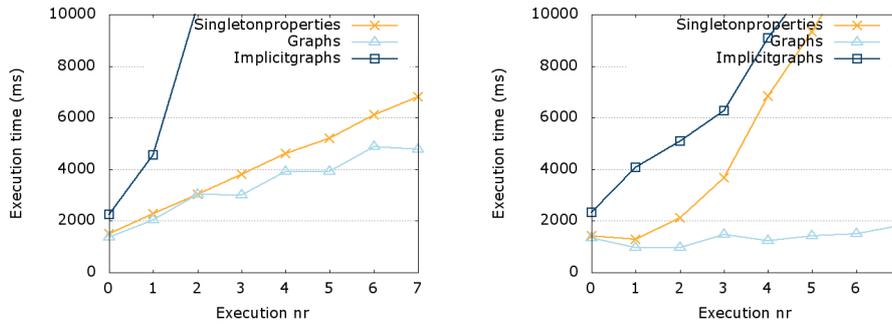
Fig. a: Time intervals without caching.

Fig. b: Time intervals with caching.
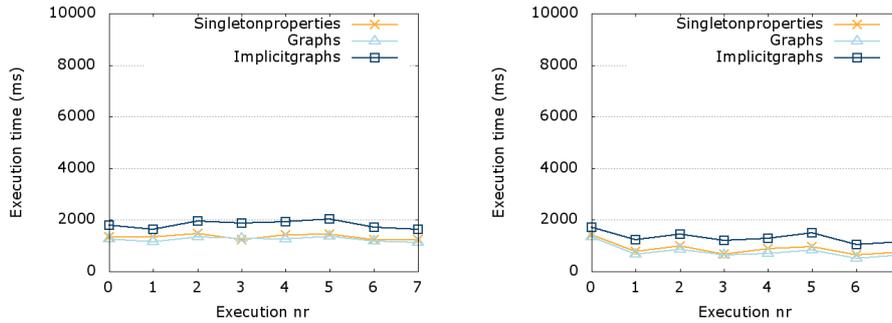
Fig. c: Expiration times without caching.

Fig. d: Expiration times with caching.

Fig. 4: Executions times for the three different types of dynamic data representation for several subsequent streaming requests. The figures show a mostly linear increase when using time intervals and constant execution times for annotation using expiration times. In general, caching results in lower execution times. They also reveal that the graph approach has the lowest execution times.

8 Conclusions

In this paper, we researched a solution for querying over dynamic data with a low server cost, by continuously polling the data based on volatility information. In this section, we draw conclusions from our evaluation results to give an answer to the research questions and hypotheses we defined in Section 3. First, the server and client costs for our solution will be compared with the alternatives. After that, the effect of our client-side cache will be explained. Next, we will discuss the effect of time annotation on the amount of requests to be sent, after which the performance of our solution will be shown and the effects of the annotation methods.

Server cost The results from Section 7 confirm Hypothesis 1, in which we wanted to know if we could lower the server cost when compared to c-sparql and cqels. Not only is the server cost for our solution more than ten times lower on average when compared to the alternatives, this cost also increases much slower for a growing number of simultaneous clients. This makes our proposed solution more scalable for the server. Another disadvantage of c-sparql and cqels is the fact that the server load for a large number of concurrent clients varies significantly, as can be seen in Figure 3a. This makes it hard to scale the required processing powers for servers using these technologies. Our solution has a low and more constant cpu usage.

Client cost The results for the client load measurements from Section 7 confirm Hypothesis 2, which stated that our solution increases the client's processing need. The required client processing power using our solution is clearly much higher than for c-sparql and cqels. This is because we redistributed the required processing power from the server to the client. In our solution, it is the client that has to do most of the work for evaluating queries, which puts less load on the server. The load on the client still remains around 5% for the largest part of the query evaluation as shown in Figure 2b. Only during the first few seconds, the query engines cpu usage peaks, which is because of the processor-intensive rewriting step that needs to be done once at the start of each dynamic query evaluation.

Caching We can also confirm Hypothesis 3 about the positive effect of caching from the results in Section 7. Our caching solution has a positive effect on the execution times. In an optimal scenario for our use case, caching would lead to an execution time reduction of 60% because three of the five triple patterns in the query for our use case from Section 4 are dynamic. For our results, this caching leads to an average reduction of 56% which is close to the optimal case. Since we are working with dynamic data, some required background-data is bound to overlap, in these cases it is advantageous to have a client-side caching solution so that these redundant requests for static data can be avoided. The longer our query evaluation runs, the more static data the cache accumulates, so the bigger the chance that there are cache hits when background data is needed in a certain query iteration. Future research should indicate what the limits of such a client-side cache for static data are, and whether or not it is advantageous to reuse this cache for different queries.

Request reduction By annotating dynamic data with a time annotation, we successfully reduced the amount of required requests for polling-based SPARQL querying to a minimum, which answers Research Question 1 about the question if clients can use volatility knowdledge to perform continuous querying. Because now, the client can derive the exact moment at which the data can change on the server, and this will be used to shedule a new query execution on the server. In future research, it is still possible to reduce the amount of requests our client engine needs to send through a better caching strategy, which could for example also temporarily cache dynamic data which changes at different frequencies. We can also look into differential data transmission by only sending data to the client that has been changed since the last time the client has requested a specific resource.

Performance For answering Research Question 2, the performance of our solution compared to alternatives, we compared our solution with two state-of-the-art approaches for dynamic SPARQL querying. Our solution significantly reduces the required server processing per client, this complexity is mostly moved to the client. This comparison shows that our technique allows data providers to offer dynamic data which can be used to continuously evaluate dynamic queries with a low server cost. Our low-cost publication technique for dynamic data is useful when the number of potential simultaneous clients is large. When this data is needed for only a small number of clients in a closed off environment and query evaluation must happen fast, traditional approaches like CQELS or C-SPARQL are advised. These are only two possible points on the *Linked Data Fragments* axis [18], depending on the publication requirements, combinations of these approaches can be used.

Annotation methods In Research Question 3, we wanted to know how the different annotation methods influenced the execution times. From the results in Section 7, we can conclude that graph-based annotation results in the lowest execution times. It can also be seen that annotation with time intervals has the problem of continuously increasing execution times, because of the continuously growing dataset. Time interval annotation can be desired if we for example want to maintain the history of certain facts, as opposed to just having the last version of facts using expiration times. In future work, we will investigate alternative techniques to support time interval annotation without the continuously increasing execution times.

In this work, the frequency at which our queries are updated is purely data-driven using time intervals or expiration times. In the future it might be interesting, to provide a control to the user to change this frequency, if for example this user only desires query updates at a lower frequency than the data actually changes.

In future work, it is important to test this approach with a larger variety of use cases. The time annotation mechanisms we use are generic enough to transform all static facts to dynamic data for any number of triples. The CityBench [1] RSP engine benchmark can for example be used to evaluate these different cases based on city sensor data. These tests must be scaled (both in terms of clients as in terms of dataset size), so that the maximum number of concurrent requests can be determined, with respect to the dataset size.

References

1. Ali, M.I., Gao, F., Mileo, A.: CityBench: A configurable benchmark to evaluate RSP engines using smart city datasets. In: The Semantic Web - ISWC 2015, Lecture Notes in Computer Science, vol. 9367, pp. 374–389 (2015)
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)
3. Barbieri, D., Braga, D., Ceri, S., Della Valle, E., Grossniklaus, M.: Stream Reasoning: Where We Got So Far. In: Proceedings of the NeFoRS2010 Workshop (2010)
4. 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)
5. 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)
6. Cyganiak, R., Wood, D., Lanthaler, M.: RDF 1.1: Concepts and abstract syntax. Recommendation, W3C (Feb 2014), http://www.w3.org/TR/2014/REC-rdf11-concepts-20140225/
7. Della Valle, E., Ceri, S., van Harmelen, F., Fensel, D.: It's a streaming world! Reasoning upon rapidly changing information. Intelligent Systems, IEEE 24(6), 83–89 (Nov 2009)
8. Gutierrez, C., Hurtado, C., Vaisman, A.: Introducing time into RDF. Knowledge and Data Engineering, IEEE Transactions on 19(2), 207–218 (Feb 2007)
9. Gutierrez, C., Hurtado, C., Vaisman, A.: Temporal RDF. In: The Semantic Web: Research and Applications, pp. 93–107 (2005)
10. iLab.t, iMinds: Virtual Wall: wired networks and applications, http://ilabt.iminds.be/virtualwall
11. Klyne, G., J. Carroll, J.: Resource Description Framework (RDF): Concepts and abstract syntax. Rec., W3C (Feb 2004), http://www.w3.org/TR/2004/REC-rdf-concepts-20040210/
12. 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)
13. Levan, C.: CQELS engine: Instructions on experimenting CQELS, https://code.google.com/p/cqels/wiki/CQELS_engine
14. Nguyen, V., Bodenreider, O., Sheth, A.: Don't like RDF reification? Making statements about statements using singleton property. In: Proceedings of the 23rd International Conference on World Wide Web. pp. 759–770. WWW '14, New York, NY, USA (2014)
15. Pérez, J., Arenas, M., Gutierrez, C.: Semantics and complexity of SPARQL. In: International semantic web conference. vol. 4273, pp. 30–43 (2006)
16. StreamReasoning: Continuous SPARQL (C-SPARQL) ready to go pack, http://streamreasoning.org/download
17. Taelman, R., Verborgh, R., Colpaert, P., Mannens, E., Van de Walle, R.: Continuously updating query results over real-time Linked Data. In: Proceedings of the 2nd Workshop on Managing the Evolution and Preservation of the Data Web (May 2016)
18. Verborgh, R., Vander Sande, M., Hartig, O., Van Herwegen, J., De Vocht, L., De Meester, B., Haesendonck, G., Colpaert, P.: Triple Pattern Fragments: a low-cost knowledge graph interface for the Web. Journal of Web Semantics 37–38, 184–206 (2016)