Continuously Updating Queries over Real-Time Linked Data

Ruben Taelman

Supervisors: Prof. dr. Erik Mannens, Prof. dr. ir. Rik Van de Walle
Counsellors: Dr. ir. Ruben Verborgh, Pieter Colpaert

Master's dissertation submitted in order to obtain the academic degree of
Master of Science in de ingenieurswetenschappen: computerwetenschappen

Department of Electronics and Information Systems
Chairman: Prof. dr. ir. Rik Van de Walle
Faculty of Engineering and Architecture
Academic year 2014-2015

Acknowledgments

First, I would like to thank Ruben Verborgh and Pieter Colpaert for their tremendous effort of aiding me in this dissertation. Their insights, feedback and honest opinions were a major help for finishing this dissertation. Their inspiring enthusiasm really made me enjoy this work.

For helping we with setting up the CQELS experiments, I'd also like to thank Chen Levan of the CQELS engine development team.

I also thank iMinds for the use of their Virtual Wall environment, together with Brecht Vermeulen who helped me with setting up my experiments in this system.

Finally, I also thank my family and friends for their support and understanding while working on my thesis.

Usage

Ruben Taelman, May 12, 2015

Continuously Updating Queries over Real-Time Linked Data

by
Ruben Taelman

Dissertation submitted for obtaining the degree of Master in Computer Science Engineering

Academic year 2014-2015

Ghent University
Faculty of Engineering
Department of Electronics and Information Systems
Chairman: Prof. dr. ir. Rik Van de Walle

Supervisors: Prof. dr. Erik Mannens, Prof. dr. ir. Rik Van de Walle
Counsellors: Dr. ir. Ruben Verborgh, Pieter Colpaert

Summary

This dissertation investigates the possibilities of having continuously updating queries over Linked Data with a focus on server availability. This work builds upon the ideas of Linked Data Fragments to let the clients do most of the work when executing a query. The server adds metadata to make the clients aware of the data volatility for making sure the query results are always up-to-date. The implementation of the framework that is proposed, is eventually tested and compared to other alternative approaches.

Samenvatting

Deze masterthesis onderzoekt de mogelijkheiden voor continue bevraving over Linked Data met een focus op de beschikbaarheid van de server. Dit werk bouwt verder op de concepten van Linked Data Fragments om de clients het meeste werk te laten doen voor de eigenlijke bevraging. De server voegt metadata toe om de clients op de hoogte te houden van de veranderbaarheid van de data om te zorgen dat de resultaten van de bevraging altijd hernieuwd worden. De implementatie van het raamwerk die hier voorgesteld is, wordt uiteindelijk getest en vergeleken met alternatieve oplossingen voor dit probleem.

Keywords: Semantic Web, Linked Data, SPARQL, continuous querying

Trefwoorden: semantisch web, Linked Data, SPARQL, continue bevraging

Continuously Updating Queries over Real-Time Linked Data

Ruben Taelman

Supervisors: prof. dr. Erik Mannens, prof. dr. ir. Rik Van de Walle, dr. ir. Ruben Verborgh, Pieter Colpaert

Abstract— **This paper investigates the possibilities of having continuously updating queries over Linked Data with a focus on server availability. This work builds upon the ideas of Linked Data Fragments to let the client do most of the work for executing the query. The server adds metadata to make clients aware of the data volatility ensuring the query results are always up-to-date. The implementation of the framework that is proposed is tested and compared to other alternative approaches.**

Keywords— **Semantic Web, Linked Data, SPARQL, continuous querying**

I. Introduction

INFORMATION is becoming more important each day. A lot of smart clients use HTTP to keep us up-to-date with everything we need to know. Some of this information requires real-time updates, like for example the information on the current train departure delays.

Linked Data [1] provides a basis for this flexible information representation and retrieval. The main problem with traditional query endpoints in this framework is low availability, which is at least partially caused by clients that can execute queries of unbounded complexity against them. *Triple Pattern Fragments* [2] provide a solution for this by moving part of the query execution to the client itself. In this work, we build upon that idea to research a solution that enables clients to execute continuously updating queries. We do this without the server needing to remember any states of the clients or pushing results to them. This requires the server to somehow *annotate* its data so that the client can efficiently determine when to pull new data.

After this section, the relevant related work will be discussed. After that, several methods for representing dynamic data will be explained, after which our complete solution is presented. Next, a use case will be explained on which the experiments are based, these results are presented thereafter. Finally, some concluding remarks will be made, together with some possible future work.

II. Related Work

A. Linked Data

The *Semantic Web* [3] is the collection of technologies for having machine-accessible data of which *Linked Data* [1, 3] is the initiative for improving these data formats for machines. This framework consists of a stack of semantic web technologies. *RDF* [4] provides a graph-based data model that is based on *triples* to structure data and interlink it. *SPARQL* [5, 6] is a language and protocol to query RDF data stores by targeting queries to SPARQL endpoints for execution. These queries are in fact graph patterns on which pattern matching is done to retrieve the matching data for the variables inside the graph patterns.

B. RDF Annotations

Metadata is used in cases where additional information about certain data is required. This metadata can be used to annotate triples [7] with temporal information, for example to indicate the data *volatility* [8].

There are several possible approaches to perform this data annotation. *a*) *Reification* [7] was the main method of annotation before RDF 1.1 [9], annotation in this way works by transforming a triple into a *reified* triple on which annotations can be added. But because of the large triple overhead *b*) *Singleton Properties* were introduced which are based on the idea of instantiating predicates so that they can be annotated. With the introduction of RDF 1.1, *c*) *Graphs* [9] could be used to annotate a context of one or more triples. The concept of graphs was standardized already by SPARQL [6] before RDF 1.1.

We investigated two different mechanisms to add time information to RDF [10, 11]: *versioning* and *time labeling*. The former requires snapshots of the complete graph to be taken every time a change in the data occurs, while the latter simply annotates triples with their change time. Time labeling is considered more extensible and introduces less overhead. A further distinction [11] was made between *point-based* and *interval-based* time labeling. Interval-based labeling is used in scenarios where multiple time ranges can each have a different value for a given fact in one dataset. Point-based labeling is used to indicate the time at which the single available fact version is valid, this can either indicate the start or end time of the validity.

A *temporal vocabulary* [10] was introduced which is capable of representing both interval-based and point-based time labeling. This vocabulary will be referred to as `tmp`.

An example of equivalent time-annotated triples using the three annotation approaches using the time vocabulary `tmp` in the *Turtle* [12] format can be found in respectively Listings 1, 2 and 3.

```
_:stmt rdf:subject :me ;
     rdf:predicate foaf:workplaceHomepage ;
     rdf:object <http://me.example.org/> ;
     tmp:interval [ tmp:initial
                 "2008-04-01T09:00:00Z"^^xsd:dateTime ;
                 tmp:final
                 "2009-11-11T17:00:00Z"^^xsd:dateTime ] .
```

LISTING 1: A time-annotated triple using reification [7] representing the valid time in an interval-based representation.

```
:me foaf:workplaceHomepage <http://me.example.org/> _:c .
_:c tmp:interval [ tmp:initial
                "2008-04-01T09:00:00Z"^^xsd:dateTime ;
               tmp:final
                "2009-11-11T17:00:00Z"^^xsd:dateTime ] .
```

LISTING 2: A time-annotated triple using graphs in the N-Quads [13] format.

```
foaf:workplaceHomepage#1 sp:singletonPropertyOf foaf:
     workplaceHomepage .
:me foaf:workplaceHomepage#1 <http://me.example.org/> .
foaf:workplaceHomepage#1 tmp:interval [
```

```
tmp:initial "2008-04-01T09:00:00Z"^^xsd:dateTime ;
tmp:final "2009-11-11T17:00:00Z"^^xsd:dateTime ] .
```

LISTING 3: A time-annotated triple using Singleton Properties [14] .

C. Stream Reasoning

Performing *continuous* queries over RDF data *streams* requires some concepts of *Stream Reasoning* [15, 16] which is in fact a version of *Stream Processing* in the context of Linked Data. This area of research integrates data streams with traditional RDF reasoners. A *window* [17] is a subsect of facts ordered by time so that not all available information has to be taken into account while reasoning. These windows can have a certain size which indicates the time range and is advanced in time by a *stepsize*. The term *Continuous Processing* [15] is used to refer to the continuous processing life cycle for continuously evaluating queries over constantly changing data as opposed to the traditional reasoning which has a specific start and endpoint. Because of this continuous processing, *Query Registration* [15, 16] 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 only evaluated once. Triples can receive timestamps. This can be done by *annotating* triples [18] with a new tuple structure that contains a timestamp $\mathcal{T}$ on which the tuple is valid. This results in a stream of monotonically non-decreasing triples. A formal representation of an RDF stream using these timestamps $\mathcal{T}$ can be found in Equation 1.

$$\begin{gathered}\dots \\ (\langle subj_i, pred_i, obj_i \rangle, \mathcal{T}_i) \\ (\langle subj_{i+1}, pred_{i+1}, obj_{i+1} \rangle, \mathcal{T}_{i+1}) \\ \dots\end{gathered} \tag{1}$$

D. SPARQL Streaming Extensions

C-SPARQL [19] is a first existing approach to querying over static and dynamic data. This system 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) [17]. The internal model of C-SPARQL internally creates queries that distribute work between the DSMS and the SPARQL engine to respectively process the dynamic and static data.

CQELS [20] is a "white box" approach, as opposed to the "black box" approaches like C-SPARQL. This means that CQELS natively implements all query operators without transforming it to another language so that it can be delegated to another system, which removes this overhead. The syntax is very similar as to that of C-SPARQL, also supporting query registration and time windows. According to previous research [20], this approach performs much better than C-SPARQL for large datasets, for simple queries and small datasets the opposite is true.

E. Triple Pattern Fragments

Experiments [21] have shown that only 30% of public endpoints reach an availability of 99%. The main cause of this low availability is the unrestricted complexity of SPARQL queries combined with the public character of SPARQL endpoints. Clients can send arbitrarily complex SPARQL queries which can form a bottleneck in endpoints. *Triple Pattern Fragments* [2] aim to solve this issue of low availability and performance of existing SPARQL endpoints. 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. The endpoints are limited to an interface in which only separate triple patterns can be queried instead of full SPARQL queries. The client is then responsible for carrying out the remaining work.

III. Dynamic Data Representation

In this work, both the interval-based and point-based time labeling are used, these are referred to as the *temporal domains*. Interval-based labeling is used to indicate the start and endpoint of the time in which triples are valid. Point-based labeling is used to indicate the expiration times of triple validity.

With expiration times, only one version of a given fact can exist in a dataset at the same time because we do not have knowledge about the start time of the validity for distinguishing multiple versions. For time intervals, multiple versions of a fact can exist. When data is very volatile, consecutive interval-based facts will accumulate quickly. If no techniques are used to aggregate or remove old data, datasets will quickly grow which can cause continuously slower query executions. This problem does not exist with expiration times because only the latest version of a fact can exist, so this volatility will not have any effect on the dataset size.

Reification, singleton properties and graphs are methods to add these expiration times or time intervals. Since our solution is built on Triple Pattern Fragments, these fragments can be used as an alternative to graphs to represent a context over triples, which we will refer to as *implicit graphs*. These implicit graphs are also different from the three alternatives in that sense that this does not require altering the structure of the original data when time annotations are added. This means that clients who do not support the retrieval of this time information can still without a problem query data annotated with implicit graphs. When reification, singleton properties or graphs-based annotation are used, these clients will not be able to retrieve this data. Table I shows an overview comparing these four methods of annotation in terms of the required amount of triples, whether or not quad or TPF support is required and if the method still allows clients who do not support time-annotated queries to retrieve the dynamic data in a static fashion.

IV. Solution

Our solution consists of an extra software layer on top of the existing Triple Pattern Fragments client. The Triple Pattern Fragments server does not require any changes, the only requirement is that the dynamic data must be time annotated with one of the possible combinations of temporal domain and method of annotation. This dynamic data should then be updated by an external process according to its temporal range, indicated by either expiration times or time intervals.

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

	Triple-count	Quads	TPF	Backwards compatible
Reification	$f_{R\_interval}(t) = 5 * t$ $f_{R\_expiration}(t) = 4 * t$ $f_{R\_interval\_better}(t) = 4 * t + 2$	no	no	no
Singleton Properties	$f_{SP\_interval}(t) = t + 3$ $f_{SP\_expiration}(t) = t + 2$	no	no	no
Explicit Graphs	$f_{EG\_interval}(t) = t + 2$ $f_{EG\_expiration}(t) = t + 1$	required	no	no
Implicit Graphs	$f_{IG\_interval}(t) = t + 2$ $f_{IG\_expiration}(t) = t + 1$	no	yes	yes

TABLE I: Overview of the most important characteristics of the different annotation types. The column *triple-count* contains the triple-count functions in terms of the original required amount of triples t. *Quads* indicates whether the annotation type requires the concept of quads. *TPF* indicates if the annotation type requires a Triple Pattern Fragments interface. The last column indicates whether or not the annotation type allows regular clients to retrieve the dynamic facts as static data.

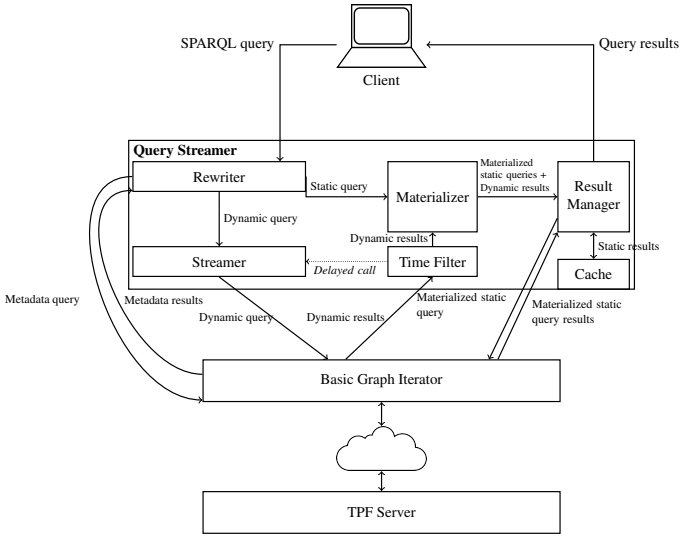


FIG. 1: Overview of the proposed architecture.

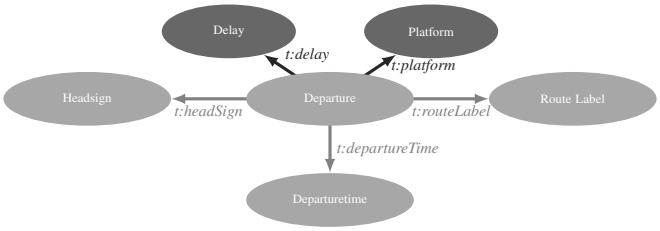


FIG. 2: Basic data model for representing train departures in one train station. The nodes in dark grey refer to dynamic data while the others are static.

The Query Streamer consists of six major components. First, there is the *Rewriter* module which is executed only once at the start of the query stream. This module is able to transform the original input query to 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 endpoint through the local TPF client. The *Streamer* module takes this dynamic query and initiates the streaming loop by executing this dynamic query and forwarding its results to the *Time Filter*. The *Time Filter* checks the time annotation for each of the results and throws out 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, this will make sure that when at least one of the results expire, a new set of results will be fetched. The filtered dynamic results will be passed on to the *Materializer* which is responsible for creating a *materialized static query*. This is a transformation of the *static query* with the dynamic results filled in. This *materialized static query* is passed to the *Result Manager* which is able to cache these queries by using the so-called *graph-connection* between the *static* and *dynamic query* as identifier. This *graph-connection* is nothing more than the intersection of all the variables in the `WHERE` clauses of the *static* and *dynamic query*. 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 query streaming.

V. USE CASE

The discussed architecture was implemented in JavaScript using Node.js [22] to allow for easy communication with the existing TPF client. We have tested our implementation with the use case of querying train departure information for a certain station. Figure 2 shows a *basic data model* for the relevant train departure information where the light grey nodes refer to static data and the dark grey nodes refer to dynamic data.

This basic data model has been adapted for each possible method of annotation. Each of these possibilities can then again be adapted to add time information for a certain temporal domain. For example when reification is used as a method of annotation, the triples `?departure t:delay ?delay` and `?departure t:platform ?platform` are reified and annotated using a temporal domain such as expiration times. A simple SPARQL query is used to retrieve all information using this basic data model, this query can be found in Listing 4. Eventually, there are eight derived data models for each possible combination of annotation and all of them are compared in terms of processing time and bandwidth usage.

```
PREFIX t: <http://example.org/train#>

SELECT DISTINCT ?delay ?headSign ?routeLabel ?platform
   ?departureTime
WHERE {
   _:id t:delay ?delay .
   _:id t:headSign ?headSign .
   _:id t:routeLabel ?routeLabel .
   _:id t:platform ?platform .
```

```
    _:id t:departureTime ?departureTime .
}
```

LISTING 4: The basic SPARQL query for retrieving all train departure information.

Next to this, we also set up an experiment to measure the client and server performance. This experiment was made up of one server and ten physical clients. Each of these clients can 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 the implementation presented is this work, C-SPARQL [19] and CQELS [20].

These tests were executed on the Virtual Wall (generation 2) [23] environment from iMinds. Each machine had two Hexa-core Intel E5645 (2.4GHz) CPU's with 24GB RAM and was running Ubuntu 12.04 LTS. For CQELS, we used the engine version 1.0.1 [24]. For C-SPARQL, this was version 0.9 [25]. 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.

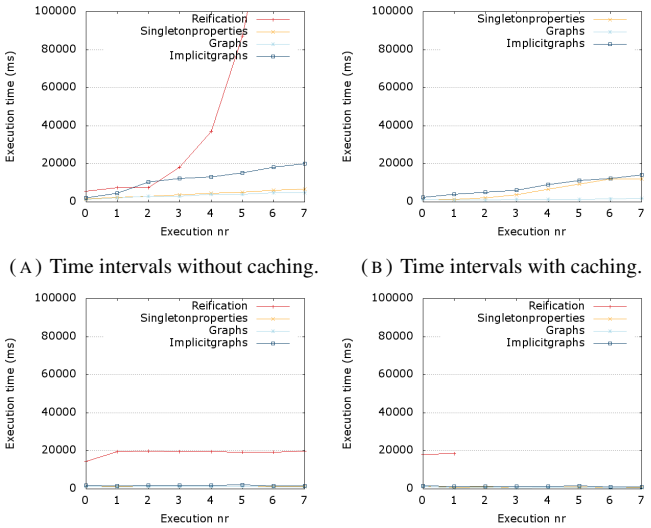
VI. RESULTS

As can be seen in Figure 3, the first experiment shows that interval-based annotation has a linear increase in execution time as the query stream executions advance, opposed to the constant execution time for when expiration times are used. This behavior was expected and this is the reason why old fact versions annotated using time intervals should not be kept indefinitely. Another observation is that reification performs much slower in all cases, this is because of the large amount of triples that is require to annotate dynamic facts. For a better comparison between singleton properties, graphs and implicit graphs using expiration times, the results from Figure 3 have been rescaled in Figure 4. It can be seen that graphs and singleton properties perform the best, with graphs being only slightly better. Implicit graphs perform a little less well than the two others, because this approach still requires some further research to be efficient in practice. It can also be noted that our caching solution has a very positive effect on the execution times. In an optimal scenario, caching would lead to a execution reduction of 60% because three of the five triple patterns in our query are dynamic. For our results, this caching lead to an average reduction of 56% (average taken without the reification results) which is very close to the optimal case.

Figure 5 shows that the primary cause of the difference in execution times is caused by the amount of data that needs to be transfered. These results closely resemble the *triple-count functions* in Table I for each method of annotation. We can conclude that the amount of triples required for annotation has the largest impact on the execution times.

A separate measurement was done for the rewriting phase, which is executed once at the start of the query streaming by the *Rewriter* module. The results in Figure 6 show that implicit graphs are significantly slower. This is because implicit graphs were defined over triple pattern fragments with determined triple values. This means that for each triple pattern, all possible values had to be checked to determine whether or not the pattern should be considered dynamic. The order of performance between the other three approaches can again be explained by the amount of required triples.

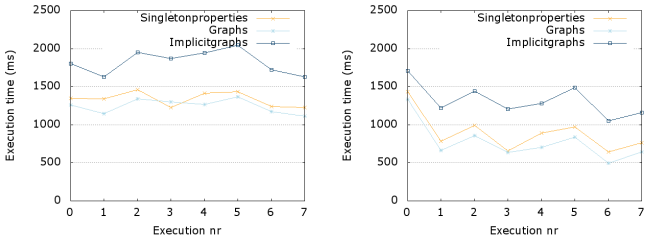
From Figures 7a and 7b we can see that our implementation significantly reduces the server load when compared to C-SPARQL and CQELS, as was the main the goal. We see that the client now pays for the largest part of the query executions, which is caused by the use of Triple Pattern Fragments. The client 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%.



(A) Time intervals without caching. (B) Time intervals with caching.

(C) Expiration times without caching. (D) Expiration times with caching.

FIG. 3: Executions times for all 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.



(A) Expiration times without caching. (B) Expiration times with caching.

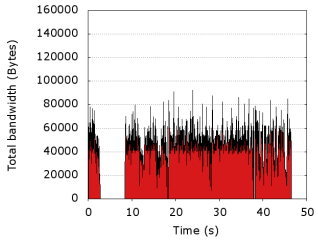
FIG. 4: Executions times for all different types of time annotation methods using expiration times for several subsequent streaming requests. These figures contain the same data as Figures 3c and 3d, but without the rapidly increasing reification results in order to reveal the other methods in more detail. They indicate the graph approach having the lowest execution times.

VII. CONCLUSION

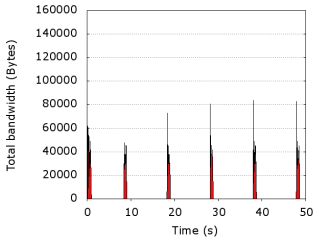
We have researched and compared different methods for continuously updating SPARQL queries, together with a solution based on Triple Pattern Fragments. Our solution proves to significantly reduce the server load at an increased bandwidth and client processing cost.

We have investigated four different methods of annotation, of which the graph-based approach proved to perform the best in our experiments. The two temporal domains, time intervals and expiration times, can be used, of which the latter is the best solution for very volatile data.

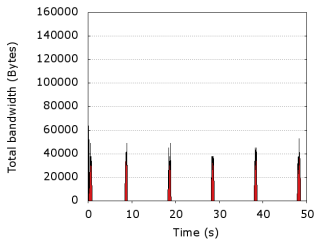
The solution presented in this work only requires an extra layer on top of the existing TPF client to enable query streaming. The data that should be seen as dynamic must be annotated with time information in its dataset.



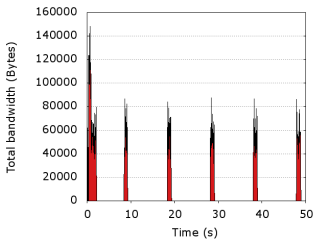
(A) The data transfer in bytes using **reification** having a total transfer of 36.46 MB over 7924 requests in this time range.



(B) The data transfer in bytes using **singleton properties** having a total transfer of 2.57 MB over 627 requests in this time range.



(C) The data transfer in bytes using **explicit graphs** having a total transfer of 2.10 MB over 557 requests in this time range.



(D) The data transfer in bytes using **implicit graphs** having a total transfer of 6.70 MB over 1191 requests in this time range.

FIG. 5: The data transfer in bytes for the four annotation methods for a duration of 50 seconds. These plots used expiration times with caching disabled. Reification uses by far the most bandwidth, while the graph approach uses the least.

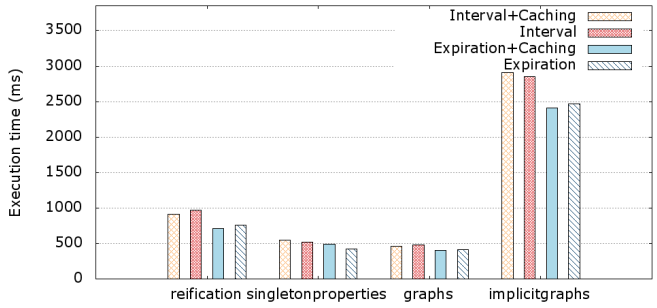
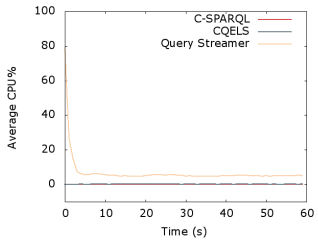
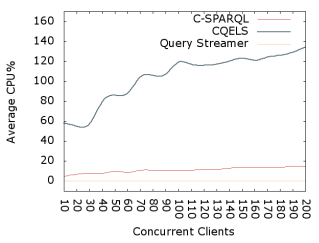


FIG. 6: Histogram of the preprocessing execution times for the different options for annotation, grouped by annotation method. The graph approach has the lowest preprocessing execution times. Expiration times are slightly faster and caching has no significant influence.



(A) Average client CPU usage for one query stream for C-SPARQL, CQELS and the solution presented in this work. Initially the CPU usage for our implementation is very high after which is converges to about 5%. The usage for C-SPARQL and CQELS is almost non-existing.



(B) Average server CPU usage for an increasing amount of clients for C-SPARQL, CQELS and the solution presented in this work. The CPU usage of this solution proves to be influenced less by the number of clients. Note that the test machine had 4 assigned cores.

FIG. 7: The client and server CPU usages for one query stream for C-SPARQL, CQELS and the solution presented in this work.

VIII. FUTURE WORK

This research is still just a first approach for enabling continuous querying using Triple Pattern Fragments and many aspects can still be improved.

- Instead of one static and dynamic query as a result of the query rewriting, *multiple queries* could be used with each a different volatility. This way, more complicated queries with different dynamic triple patterns can more efficiently be cached and queried.
- Even though the graph-based method of annotation proved to be the most efficient in these experiments, *implicit graphs* could still be improved on many levels to become even more efficient.
- In this work, we have assumed that the change-times of the dynamic data was known by the data provider. *Determining expiration times and time intervals* might become quite complex and could require advanced pattern matching algorithms on the data change history.
- If the TPF client could efficiently add `FILTER`-support, the expiration times and time intervals could become much more efficient to lookup. This could however increase the server load again.
- As was presented in a similar research [26], *static background data* might not remain the same forever. So the client *Caching* module should take this into account.
- The results presented here might differ a lot with other use cases, so similar tests for other query types could produce interesting results. Queries that request more static data would for example become relatively more efficient.
- If a larger testing environment would be available, these server and client performance experiments should be executed for a much larger amount of concurrent clients to determine the actual limits of the server using this approach.

REFERENCES

[1] Tim Berners-Lee. Linked Data, July 2006. URL `http://www.w3.org/DesignIssues/LinkedData.html`

[2] Ruben Verborgh, Miel Vander Sande, Pieter Colpaert, Sam Coppens, Erik Mannens, and Rik Van de Walle. Web-Scale Querying through Linked Data Fragments. In *Proceedings of the 7th Workshop on Linked Data on the Web*. April 2014. URL `http://events.linkeddata.org/ldow2014/papers/ldow2014_paper_04.pdf`

[3] Tim Berners-Lee, James Hendler, Ora Lassila, et al. The Semantic Web. *Scientific american*, 284(5):28–37, 2001. URL `http://isel2918929391.googlecode.com/svn-history/r347/trunk/RPC/Slides/p01_theSemanticWeb.pdf`

[4] JJ Carol and G Klyne. Resource Description Framework: Concepts and Abstract Syntax. Recommendation, W3C, February 2014. URL `http://www.w3.org/TR/2014/REC-rdf11-concepts-20140225/`

[5] Eric PrudHommeaux, Andy Seaborne, et al. SPARQL Query Language for RDF. *W3C recommendation*, 15, 2008. URL `http://www.w3.org/TR/rdf-sparql-query/`

[6] Steve Harris, Andy Seaborne, and Eric Prud'hommeaux. SPARQL 1.1 Query Language. Recommendation, W3C,

March 2013.
URL http://www.w3.org/TR/2013/REC-sparql11-query-20130321/

[7] Nuno Lopes, Antoine Zimmermann, Aidan Hogan, Gergely Lukácsy, Axel Polleres, Umberto Straccia, and Stefan Decker. RDF Needs Annotations. In *W3C Workshop on RDF Next Steps, Stanford, Palo Alto, CA, USA*. Citeseer, 2010.
URL http://www.w3.org/2009/12/rdf-ws/papers/ws09

[8] Donald Ballou, Richard Wang, Harold Pazer, and Giri Kumar.Tayi. Modeling Information Manufacturing Systems to Determine Information Product Quality. *Management Science*, 44(4):pp. 462–484, 1998. ISSN 00251909.
URL http://www.jstor.org/stable/2634609

[9] Frank Manola, Eric Miller, and Brian McBride. RDF 1.1 Primer. Working group note, W3C, June 2014.
URL http://www.w3.org/TR/2014/NOTE-rdf11-primer-20140624/#section-turtle-family

[10] C. Gutierrez, C.A Hurtado, and A Vaisman. Introducing Time into RDF. *Knowledge and Data Engineering, IEEE Transactions on*, 19(2):207–218, Feb 2007. ISSN 1041-4347. doi:10.1109/TKDE.2007.34.
URL http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4039284

[11] Claudio Gutierrez, Carlos Hurtado, and Alejandro Vaisman. Temporal RDF. In *The Semantic Web: Research and Applications*, pages 93–107. Springer, 2005.
URL http://link.springer.com/chapter/10.1007/11431053_7

[12] David Beckett, Tim Berners-Lee, Eric Prudhommeaux, and Gavin Carothers. Turtle-terse RDF triple language. Recommendation, W3C, February 2014.
URL http://www.w3.org/TR/2014/REC-turtle-20140225/

[13] Gavin Carothers. RDF 1.1 N-Quads: A line-based syntax for RDF datasets. Recommendation, W3C, February 2014.
URL http://www.w3.org/TR/2014/REC-n-quads-20140225/

[14] Vinh Nguyen, Olivier Bodenreider, and Amit Sheth. Don't Like RDF Reification? Making Statements About Statements Using Singleton Property. In *Proceedings of the 23rd International Conference on World Wide Web*, WWW '14, pages 759–770. ACM, New York, NY, USA, 2014. ISBN 978-1-4503-2744-2. doi:10.1145/2566486.2567973.
URL http://doi.acm.org/10.1145/2566486.2567973

[15] E. Della Valle, S. Ceri, F. van Harmelen, and D. Fensel. It's a Streaming World! Reasoning upon Rapidly Changing Information. *Intelligent Systems, IEEE*, 24(6):83–89, Nov 2009. ISSN 1541-1672. doi:10.1109/MIS.2009.125.
URL http://www.few.vu.nl/~frankh/postscript/IEEE-IS09.pdf

[16] Davide Barbieri, Daniele Braga, Stefano Ceri, Emanuele Della Valle, and Michael Grossniklaus. Stream Reasoning: Where We Got So Far. In *Proceedings of the NeFoRS2010 Workshop, co-located with ESWC2010*. 2010.
URL http://wasp.cs.vu.nl/larkc/nefors10/paper/nefors10_paper_0.pdf

[17] Arvind Arasu, Brian Babcock, Shivnath Babu, John Cieslewicz, Mayur Datar, Keith Ito, Rajeev Motwani, Utkarsh Srivastava, and Jennifer Widom. STREAM: The Stanford Data Stream Management System. *Book chapter*, 2004.
URL http://ilpubs.stanford.edu:8090/641/1/2004-20.pdf

[18] Davide Francesco Barbieri, Daniele Braga, Stefano Ceri, Emanuele Della Valle, and Michael Grossniklaus. Querying RDF Streams with C-SPARQL. *SIGMOD Rec.*, 39(1):20–26, September 2010. ISSN 0163-5808. doi:10.1145/1860702.1860705.
URL http://doi.acm.org/10.1145/1860702.1860705

[19] Davide Francesco Barbieri, Daniele Braga, Stefano Ceri, and Michael Grossniklaus. An Execution Environment for C-SPARQL Queries. In *Proceedings of the 13th International Conference on Extending Database Technology*, EDBT '10, pages 441–452. ACM, New York, NY, USA, 2010. ISBN 978-1-60558-945-9. doi:10.1145/1739041.1739095.
URL http://doi.acm.org/10.1145/1739041.1739095

[20] Danh Le-Phuoc, Minh Dao-Tran, Josiane Xavier Parreira, and Manfred Hauswirth. A Native and Adaptive Approach for Unified Processing of Linked Streams and Linked Data. In *The Semantic Web–ISWC 2011*, pages 370–388. Springer, 2011.
URL http://iswc2011.semanticweb.org/fileadmin/iswc/Papers/Research_Paper/05/70310368.pdf

[21] Carlos Buil-Aranda, Aidan Hogan, Jürgen Umbrich, and Pierre-Yves Vandenbussche. SPARQL Web-Querying Infrastructure: Ready for Action? In *The Semantic Web–ISWC 2013*, pages 277–293. Springer, 2013.
URL http://link.springer.com/chapter/10.1007/978-3-642-41338-4_18

[22] Joycent, Inc. Node.js.
URL http://nodejs.org/

[23] iMinds iLab.t. Virtual Wall: wired networks and applications.
URL http://ilabt.iminds.be/virtualwall

[24] Chen Levan. CQELS engine: Instructions on experimenting CQELS.
URL https://code.google.com/p/cqels/wiki/CQELS_engine

[25] StreamReasoning. Continuous SPARQL (C-SPARQL) Ready To Go Pack.
URL http://streamreasoning.org/download

[26] Soheila Dehghanzadeh, Daniele Dell'Aglio, Shen Gao, Emanuele Della Valle, Alessandra Mileo, and Abraham Bernstein. Approximate Continuous Query Answering Over Streams and Dynamic Linked Data Sets. In *15th International Conference on Web Engineering*. s.n., Switzerland, June 2015.
URL http://dx.doi.org/10.5167/uzh-110296

Continu actualiserende bevraging van Linked Data

Ruben Taelman

Supervisors: prof. dr. Erik Mannens, prof. dr. ir. Rik Van de Walle, dr. ir. Ruben Verborgh, Pieter Colpaert

Abstract—**Deze paper onderzoekt de mogelijkheiden voor continue bevraving over Linked Data met een focus op de beschikbaarheid van de server. Dit werk bouwt verder op de concepten van Linked Data Fragments om de clients het meeste werk te laten doen voor de eigenlijke bevraging. De server voegt metadata toe om de clients op de hoogte te houden van de veranderbaarheid van de data om te zorgen dat de resultaten van de bevraging altijd hernieuwd worden. De implementatie van het raamwerk die hier voorgesteld is, wordt uiteindelijk getest en vergeleken met alternatieve oplossingen voor dit probleem.**

Keywords— **semantisch web, Linked Data, SPARQL, continue bevraging**

I. Introductie

INFORMATIE wordt elke dag belangrijker. Veel intelligente clients gebruiken HTTP om ons up-to-date te houden over alles wat we nodig hebben. Delen van deze informatie verwachten updates in werkelijke tijd, net als bijvoorbeeld de informatie over de huidige trein vertragingen.

Linked Data [1] voorziet een basis voor deze flexibele manier van data representatie en bevraging. Het grootste probleem met de traditionele query eindpunten in dit raamwerk is de lage beschikbaarheid, deze is ten minste deels veroorzaakt door clients die queries van onbeperkte complexiteit hierop kunnen uitvoeren. *Triple Pattern Fragments* [2] bieden een oplossing voor dit probleem door een deel van de query uitvoering te verplaatsen naar de client-zijde. In dit werk bouwen we verder op dit idee om een oplossing te vinden die clients toelaat om continu actualiserende queries uit te voeren. We doen dit zonder dat de server enige toestand van de clients moet onthouden of zelf volledige resultaten moet sturen. Hiervoor moet de server op een of andere manier zijn data *annoteren* zodat de client op een efficiënte manier kan bepalen wanneer nieuwe resultaten opgevraagd moeten worden.

Na deze sectie worden enkele relevante onderzoeken besproken. Hierna worden enkele methoden uitgelegd waarmee dynamische gegevens kunnen worden voorgesteld, waarna onze volledige oplossing wordt uitgelegd. Als volgende wordt een use case voorgesteld waarop onze experimenten gebaseerd zijn, waarvan de resultaten hierna worden gepresenteerd. Tenslotte worden enkele besluitende opmerkingen gemaakt, samen met mogelijke opvolgende onderzoeksmogelijkheden.

II. Gerelateerd werk

A. Linked Data

Het *Semantisch Web* [3] is een collectie van technologiën om data leesbaar te maken voor machines waarvoor *Linked Data* [1, 3] het initiatief is om deze gegevensrepresentatie te verbeteren. Dit raamwerk bevat enkele verschillende semantisch web technologiën. *RDF* [4] biedt een data model aan die gebaseerd is op grafen en *triples* gebruikt om data te structureren en deze naar elkaar te linken. *SPARQL* [5, 6] is een taal en protocol om RDF data opslagplaatsen te bevragen door middel van SPARQL eindpunten die deze kunnen uitvoeren. Deze queries zijn graaf patronen waarop patroonherkenning wordt toegepast om passende data voor de variabelen in die graaf op te vragen.

B. RDF Annotatie

Metadata wordt gebruikt in gevallen waar extra informatie nodig is over de data. Deze metadata kan worden gebruikt om triples te annoteren [7] met temporele informatie, bijvoorbeeld om de *volatiliteit* [8] van data aan te duiden.

Er zijn verschillende mogelijkheden om deze data annotatie uit te voeren. *a*) *Reification* [7] was de belangrijkste annotatie methode voor RDF 1.1 [9], annotatie op deze manier wordt gedaan door een triple om te zetten naar een *reified* triple waarop annotaties kunnen toegevoegd worden. Door het grote aantal triples die hiervoor nodig zijn werden *b*) *Singleton Properties* geïntroduceerd waarmee predikaten geïnstantieerd worden zodat hieraan annotaties kunnen worden toegevoegd. Met de introductie van RDF 1.1 konden *c*) *Graphs* [9] gebruikt worden om een context van één of meer triples te annoteren. Dit concept was al reeds gestandaardiseerd door SPARQL [6] voor RDF 1.1.

We hebben twee verschillende mechanismen onderzocht om tijdsinformatie toe te voegen in RDF [10, 11]: *graaf versies* en *tijd labeling*. De eerste werkt op basis van momentopnames van de volledige graaf die genomen worden elke keer een verandering optreedt in de data, terwijl de tweede enkel de triples annoteert met hun tijdstip van aanpassing. Tijd labeling wordt gezien als beter uitbreidbaar en efficiënter. Een verdere opdeling [11] werd gemaakt tussen *punt-gebaseerde* end *interval-gebaseerde* tijd labeling. Interval-gebaseerde labeling wordt gebruikt in scenarios waar verschillende waarden voor een bepaald feit kunnen bestaan op verschillende tijdstippen in een dataset. Punt-gebaseerde labeling wordt gebruikt om de tijd aan te duiden waarop een unieke versie van een feit beschikbaar is, dit kan verwijzen naar de start- of eindtijd van de geldigheidsduur.

Een *temporeel vocabularium* [10] werd geïntroduceerd om zowel interval-gebaseerde als punt-gebaseerde tijd labeling voor te stellen. Er wordt naar dit vocabularium verwezen als `tmp`.

Een voorbeeld van equivalente tijd geannoteerde triples gebruikmakende van de drie verschillende methoden van annotatie in het tijd vocabularium `tmp` in het *Turtle* [12] formaat kan gevonden worden in respectievelijk Listings 1, 2 and 3.

```
_:stmt rdf:subject :me ;
     rdf:predicate foaf:workplaceHomepage ;
     rdf:object <http://me.example.org/> ;
     tmp:interval [ tmp:initial
                 "2008-04-01T09:00:00Z"^^xsd:dateTime ;
                 tmp:final
                 "2009-11-11T17:00:00Z"^^xsd:dateTime ] .
```

LISTING 1: Een tijd geannoteerde triple gebruikmakende van reification [7] die de geldigheid tijd voorstel door middel van interval-gebaseerde labeling.

```
:me foaf:workplaceHomepage <http://me.example.org/> _:c .
_:c tmp:interval [ tmp:initial
                "2008-04-01T09:00:00Z"^^xsd:dateTime ;
               tmp:final
                "2009-11-11T17:00:00Z"^^xsd:dateTime ] .
```

LISTING 2: Een tijd geannoteerde triple gebruikmakende van graphs in het N-Quads [13] formaat.

```
foaf:workplaceHomepage#1 sp:singletonPropertyOf foaf:
     workplaceHomepage .
:me foaf:workplaceHomepage#1 <http://me.example.org/> .
foaf:workplaceHomepage#1 tmp:interval [
   tmp:initial "2008-04-01T09:00:00Z"^^xsd:dateTime ;
   tmp:final "2009-11-11T17:00:00Z"^^xsd:dateTime ] .
```

LISTING 3: Een tijd geannoteerde triple gebruikmakende van Singleton Properties [14] .

C. Stream Redenering

Om *continue* queries over RDF data *streams* te kunnen uitvoeren hebben we nood aan enkele concepten van *Stream Redenering* [15, 16] die eigenlijk een versie is van *Stream Verwerking* in de context van Linked Data. Dit domein van onderzoek integreert data streams met traditionele RDF redenering. Een *venster* [17] is een beperkte verzameling van feiten die geordend is in tijd zodat niet alle beschikbare informatie moet bekeken worden tijdens het redeneren. Deze vensters hebben een zekere lengte die de tijdsduur aanduidt en vordert in tijd op basis van een *stapgrootte*. De term *Continue Verwerking* [15] wordt gebruikt om te verwijzen naar de cyclus van continue uitvoering voor continu uitvoerende queries over veranderende data, wat verschilt van de traditionele verwerking omdat deze een specifiek start- en eindpunt heeft. Door deze continue verwerking doen clients aan *Query Registratie* [15, 16] om ervoor te zorgen dat het SPARQL eindpunt deze query herhalend kan uitvoeren, in tegenstelling tot traditionele eindpunten waarbij de query eenmalig wordt uitgevoerd. Triples krijgen een tijdstip. Dit kan gebeuren door de *annotatie* van triples [18] met een nieuw triple structuur die de tijdstip $\mathcal{T}$ bevat en de tijd aanduidt waarop dit tuple geldig is. Dit resulteert in een stream van monotoon niet-afnemende triples in tijd. Een formele representatie van een RDF stream gebruikmakende van deze tijdstippen $\mathcal{T}$ kan gevonden worden in Vergelijking 1.

$$\begin{gathered}\dots \\ (\langle subj_i, pred_i, obj_i\rangle, \mathcal{T}_i) \\ (\langle subj_{i+1}, pred_{i+1}, obj_{i+1}\rangle, \mathcal{T}_{i+1}) \\ \dots\end{gathered} \tag{1}$$

D. SPARQL Streaming Extensies

C-SPARQL [19] is een eerste benadering tot de bevraging over statische en dynamische gegevens. Dit systeem laat de client toe om een query te registreren in een uitgebreide SPARQL syntaxis die toelaat om vensters te definiëren over dynamische data. De uitvoering van queries is gebaseerd op de combinatie van traditionele SPARQL verwerking met een *Data Stream Management System* (DSMS) [17]. Het interne model van C-SPARQL maakt intern queries die het werk verdelen over het DSMS en de SPARQL verwerker om respectievelijk de dynamische en statische data te behandelen.

CQELS [20] is een „witte doos" benadering, in tegenstelling tot de „zwarte doos" benaderingen zoals C-SPARQL. Dit wilt zeggen dat CQELS zelf volledig alle query operatoren implementeert zonder dat een omzetting naar een andere taal en delegatie naar een ander systeem nodig is, wat het proces efficiënter kan maken. De syntaxis lijkt goed op die van C-SPARQL, deze ondersteunt ook query registratie en tijd vensters. Volgens eerder onderzoek [20] werkt deze benadering veel beter dan C-SPARQL voor grote datasets, voor eenvoudige queries en kleine datasets geldt het tegenovergestelde.

E. Triple Pattern Fragments

Experimenten [21] hebben aangetoond dat slechts 30% van de publieke eindpunten een beschikbaarheid hebben van tenminste 99%. De grootste oorzaak van deze lage beschikbaarheid is de onbeperke complexiteit van SPARQL queries gecombineerd met het publieke karakter van SPARQL eindpunten. Clients kunnen SPARQL queries van arbitraire complexiteit sturen wat zeer zwaar is voor eindpunten. *Triple Pattern Fragments* [2] probeert dit probleem van lage beschikbaarheid en performantie op te lossen. Dit gebeurt door een deel van de query verwerking te verplaatsen naar die client, wat de server belasting verlaagt ten koste van een toename in data overdracht. De eindpunten zijn beperkt tot een interface waartegen enkel aparte triple patronen kunnen worden bevraagd in plaats van volledige SPARQL queries. De client is dan zelf verantwoordelijk om de rest van de query uitvoering te doen.

III. DYNAMISCHE DATA REPRESENTATIE

In dit werk wordt zowel de interval-gebaseerde als de punt-gebaseerde tijd labeling gebruikt, hier wordt verder naartoe verwezen als de *temporele domeinen*. Interval-gebaseerde labeling wordt gebruikt om een start- en eindpunt in tijd aan te duiden waartussen triples geldig zijn. Punt-gebaseerde labeling wordt gebruikt om de vervaltijden van triples voor te stellen.

Met vervaltijden kan er slechts één versie van een feit bestaan in een dataset omdat hier anders niet genoeg kennis over is om de starttijd van de geldigheid aan te duiden. Voor tijd intervallen kunnen meerdere versie van een feit bestaan. Wanneer data zeer volatiel is kunnen opeenvolgende interval-gebaseerde feiten snel opstapelen. Indien geen technieken gebruikt worden om oude data te aggregeren of te verwijderen, zullen datasets snel groeien wat zal leiden tot continu trager uitvoeren queries. Dit probleem bestaat niet bij vervaltijden omdat daar enkel de laatste versie van een feit kan bestaan, dus de volatiliteit zal hier geen effect hebben op de grootte van de dataset.

Reification, singleton properties en graphs zijn methoden om vervaltijden of tijd intervallen toe te voegen. Aangezien onze oplossing gebaseerd is op Triple Pattern Fragments, kunnen deze fragmenten gebruikt worden als alternatief voor graphs om een context over triples voor te stellen, we zullen hier naartoe verwijzen als *implicit graphs*. Deze implicit graphs zijn ook verschillend van de drie alternatieven op het vlak dat hiermee de structuur van de originele data niet aangepast moet worden wanneer annotaties worden toegevoegd. Dit betekent dat clients die deze tijd annotaties niet ondersteunen nog steeds zonder problemen deze ruwe data kunnen bevragen. Wanneer reification, singleton properties of graphs-gebaseerde annotatie gebruikt worden, zullen deze clients niet meer in staat zijn om deze data op te vragen. Tabel I toont een overzicht die deze vier methoden van annotatie vergelijkt in termen van het benodigde aatal triples, indien quad of TPF ondersteuning nodig is en of de methode clients die deze annotatie niet ondersteunen nog steeds toelaat om de dynamische data op een statische manier op te vragen.

IV. OPLOSSING

Onze oplossing is opgebouwd uit een extra software laag bovenop de bestaande Triple Pattern Fragments client. De Triple Pattern Fragments server heeft geen veranderingen nodig, behalve dat de dynamische data hierin moet geannoteerd zijn op basis van een mogelijke combinatie van temporeel domein en

	Aantal triples	Quads	TPF	Ondersteuning voor traditionele clients
Reification	$f_{R\_interval}(t) = 5 * t$ $f_{R\_vervaltijd}(t) = 4 * t$ $f_{R\_interval\_beter}(t) = 4 * t + 2$	nee	nee	nee
Singleton Properties	$f_{SP\_interval}(t) = t + 3$ $f_{SP\_vervaltijd}(t) = t + 2$	nee	nee	nee
Explicit Graphs	$f_{EG\_interval}(t) = t + 2$ $f_{EG\_vervaltijd}(t) = t + 1$	noodzakelijk	nee	nee
Implicit Graphs	$f_{IG\_interval}(t) = t + 2$ $f_{IG\_vervaltijd}(t) = t + 1$	nee	ja	ja

TABLE I: Overzicht van de meest belangrijke eigenschappen van de verschillende manieren van annotatie. De kolom *aantal triples* bevat functies die het aantal triples aanduidt in termen van het originele aantal triples t. *Quads* toont aan indien dit type van annotatie het concept van quads nodig heeft. *TPF* duidt aan als het type een Triple Pattern Fragments interface nodig heeft. De laatste kolom toont welke types toelaten om gewone clients de dynamische feiten als statische data op te vragen.

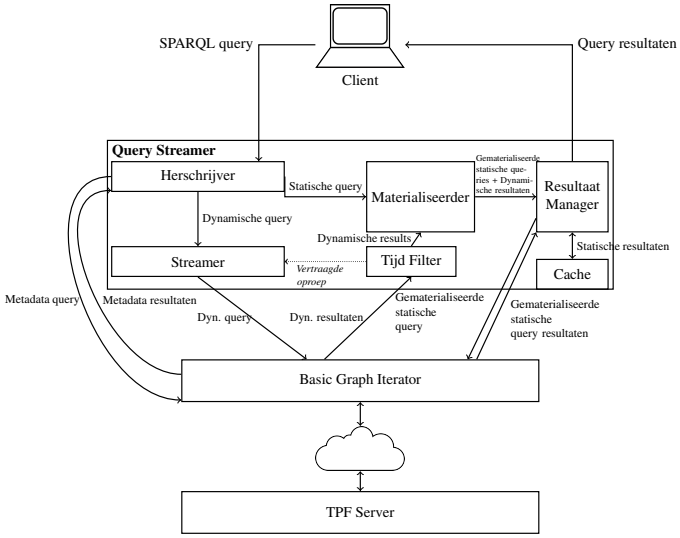


FIG. 1: Overzicht van de voorgestelde architectuur.

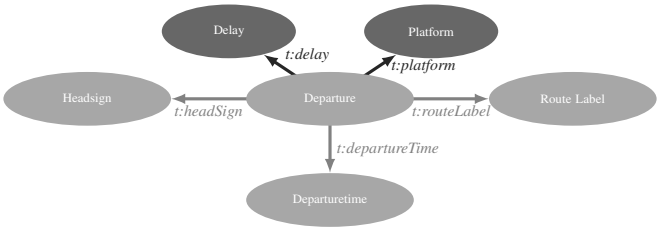


FIG. 2: Basis data model voor de representatie van trein informatie in een station. De lichtgrijze punten verwijzen naar statische data en de donkergrijze punten verwijzen naar dynamische data.

annotatie methode. Deze dynamische data moet dan geüpdatet worden door een extern proces op basis van de temporele duur, voorgesteld door vervaltijden op tijd intervallen.

Figuur 1 toont een overzicht van de architectuur voor deze extra laag bovenop de TPF client, wat vanaf nu de *Query Streamer* genoemd zal worden. Bovenaan het diagram is de client te zien die een gewone SPARQL query naar de Query Streamer stuurt en een stream van query resultaten terug krijgt. De Query Streamer kan queries uitvoeren door de lokale *Basic Graph Iterator* die deel uitmaakt van de TPF client en zelf queries kan uitvoeren op een TPF server.

De Query Streamer is opgebouwd uit zes belangrijke componenten. Eerst is er de *Herschrijver* module die éénmalig wordt uitgevoerd bij de start van de query stream. Deze module is in staat om de originele query om te vormen naar een *statische* en *dynamische* query die respectievelijk de statische achtergrond gegevens en de tijd geannoteerde veranderende data kunnen opvragen. Deze transformatie gebeurt op basis van de verkregen informatie door de bevraging van metadata over de triple patronen op het eindpunt door de lokale TPF client. De *Streamer* module neemt deze dynamische query en initialiseert de stream lus door deze dynamische query uit te voeren en zijn resultaten door te sturen naar de *Tijd Filter*. De *Tijd Filter* bekijkt alle tijd annotaties en verwijdert deze die niet geldig zijn voor het huidige tijdstip. De minimale vervaltijd van alle resultaten wordt dan bepaald en gebruikt als vertraagde oproep naar de *Streamer* module, wat ervoor zal zorgen dat wanneer tenminste één van de resultaten verloopt, een nieuwe set resultaten opgevraagd zal worden. De gefilterde dynamische resultaten worden dan doorgegeven aan de *Materialiseerder* die verantwoordelijk is om een *gematerialiseerde statische query* te maken. Dit is een transformatie van de *statische query* waarin de dynamische resultaten ingevuld zijn. Deze *gematerialiseerde statische query* wordt doorgegeven aan de *Resultaat Manager* die in staat is om deze queries te cachen door gebruik te maken van de zogenaamde *graaf-verbinding* tussen de *statische* en *dynamische query* als identificator. Deze *graaf-verbinding* is niets meer dan de doorsnede van alle variabelen in de `WHERE` clausule van de *statische* en *dynamische query*. Tenslotte haalt de *Resultaat Manager* de vorige *gematerialiseerde statische query* resultaten van de lokale cache of voert deze query uit voor de eerste keer en slaat deze op in de cache. De resultaten worden dan naar de client gestuurd die de query stream had geïnitialiseerd.

V. USE CASE

De besproken architectuur is geïmplementeerd in Javascript gebruikmakende van Node.js [22] om te zorgen voor eenvoudige communicatie met de bestaande TPF client. We hebben onze implementatie getest op de bevraging van de trein informatie van één bepaald station. Figuur 2 toont een *basis data model* voor de relevante trein informatie waarin de lichtgrijze punten verwijzen naar statische data en de donkergrijze punten verwijzen naar dynamische data.

Dit basis data model is aangepast voor elke mogelijke manier van annotatie. Elk van deze mogelijkheden kan dan nog eens worden aangepast om tijd informatie toe te voegen voor

een bepaald temporeel domein. Wanneer bijvoorbeeld reification gebruikt wordt als manier van annotatie worden de triples `?departure t:delay ?delay` en `?departure t:platform ?platform` omgevormd en geannoteerd met een temporeel domein zoals bijvoorbeeld vervaltijden. Een eenvoudige SPARQL query wordt gebruikt om alle informatie op te vragen voor dit basis data model, deze query is te vinden in Listing 4. Uiteindelijk zijn er acht afgeleide data modellen voor elke mogelijke combinatie van annotatie en deze worden allemaal vergeleken in termen van uitvoeringstijd en hoeveelheid data er wordt verstuurd.

```
PREFIX t: <http://example.org/train#>

SELECT DISTINCT ?delay ?headSign ?routeLabel ?platform
   ?departureTime
WHERE {
   _:id t:delay ?delay .
   _:id t:headSign ?headSign .
   _:id t:routeLabel ?routeLabel .
   _:id t:platform ?platform .
   _:id t:departureTime ?departureTime .
}
```

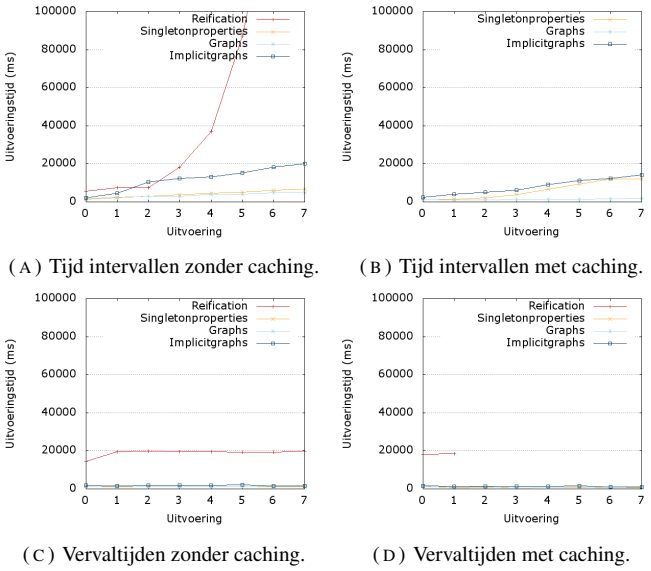
LISTING 4: De basis SPARQL query om alle trein informatie op te halen.

Hiernaast hebben we ook een experiment opgezet om de client en server performantie te meten. Dit experiment was opgebouwd uit een enkele server en tien fysieke clients. Elk van deze clients kan één tot tien simultane unieke queries uitvoeren. In totaal hebben we dus een reeks van 10 tot 200 simultane query uitvoeringen. Deze opzet werd gebruikt om de client en server performantie te testen van de implementatie die voorgesteld werd in dit werk, C-SPARQL [19] en CQELS [20].

Deze testen werden uitgevoerd op de Virtual Wall (generatie 2) [23] omgeving van iMinds. Elke machine had twee Hexacore Intel E5645 (2.4GHz) CPU's met 24GB RAM en liep op Ubuntu 12.04 LTS. Voor CQELS gebruikten we hun applicatie versie 1.0.1 [24]. Voor C-SPARQL was dit versie 0.9 [25]. De dataset hiervoor had ongeveer 300 statische triples en ongeveerd 200 dynamische triples die elke tien seconden aangemaakt en verwijderd werden.

VI. RESULTATEN

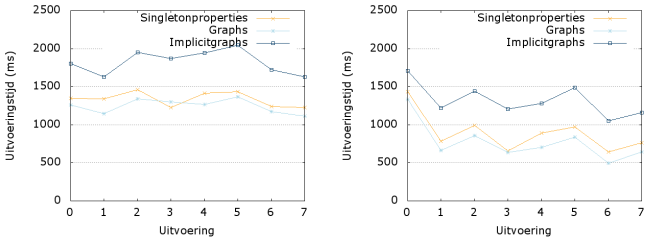
Zoals te zien is in Figuur 3 toont het eerste experiment dat interval-gebaseerde annotatie een lineaire toename heeft in uitvoeringstijd wanneer de query stream vordert, in tegenstelling tot de constante uitvoeringstijd voor wanneer vervaltijden gebruikt worden. Dit gedrag werd verwacht en dit is de reden waarom oude versies van feiten beter niet voor eeuwig in de dataset gehouden worden. Een andere observatie is dat reification veel trager is in alle gevallen, dit is door het grote aantal triples die nodig is voor dit type van annotatie. Om een betere vergelijking te kunnen maken tussen singleton properties, graphs and implicit graphs gebruikmakende van vervaltijden, werden de resultaten van Figuur 3 herschaald in Figuur 4. Hier is te zien dat graphs and singleton properties het beste presteren, met graphs die net iets beter zijn. Implicit graphs presteren iets slechter, voornamelijk omdat deze benadering verder onderzoek nodig heeft om in de praktijk bruikbaar te zijn. Er kan ook opgemerkt worden dat onze caching oplossing een zeer positief effect heeft op de uitvoeringstijden. In het optimale scenario zou caching leiden tot een uitvoeringstijd reductie van 60% omdat drie van de vijf triple patronen in onze query dynamisch zijn. In onze resultaten leidt caching tot een gemiddelde reductie van 56% (gemiddelde genomen zonder resultaten van reification), wat zeer dicht ligt bij het optimale geval.



(A) Tijd intervallen zonder caching. (B) Tijd intervallen met caching.

(C) Vervaltijden zonder caching. (D) Vervaltijden met caching.

FIG. 3: Uitvoeringstijden voor alle verschillende types van dynamische data voorstelling voor verschillende opeenvolgende stream bevragingen. De figuren tonen een ongeveer lineaire toename bij tijd intervallen en constante uitvoeringstijden voor annotatie met vervaltijden.



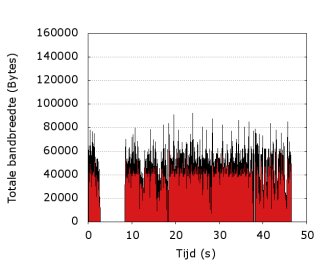
(A) Vervaltijden zonder caching. (B) Vervaltijden met caching.

FIG. 4: Uitvoeringstijden voor alle verschillende types van dynamische data voorstelling voor verschillende opeenvolgende stream bevragingen. De figuren bevatten dezelfde data als Figuren 3c en 3d, maar zonder de snel toenemende resultaten voor reification zodat de resultaten voor de andere methoden in meer detail zichtbaar worden. Hier is op te zien dat de graph manier de laagste uitvoeringstijden heeft.

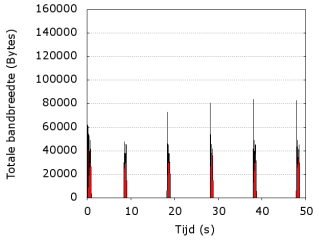
Figuur 5 toont dat de belangrijkste oorzaak van het verschil in uitvoeringstijden veroorzaakt wordt door de hoeveelheid data die moet doorgestuurd worden. Deze resultaten lijken goed op de *triple functies* in Table I voor elke methode van annotatie. We kunnen besluiten dat het aantal benodigde triples voor annotatie de grootste impact heeft op de uitvoeringstijden.

Een aparte meting is gedaan voor de herschrijf fase, die wordt uitgevoerd aan de start van de query stream door de *Herschrijver* module. De resultaten in Figuur 6 tonen dat implicit graphs veel trager zijn. Dit komt doordat implicit graphs gedefinieerd zijn over triple patroon fragmenten met gedetermineerde triple waarden. Dit wilt zeggen dat voor elk triple patroon, alle mogelijk waarden moeten gecontroleerd worden of deze wel of niet als dynamisch moeten gezien worden. De orde van performantie tussen de drie andere benaderingen kan weer verklaard worden door het aantal benodigde triples.

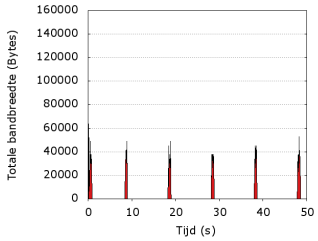
In Figuren 7a en 7b kunnen we zien dat onze implementatie de server belasting significant verlaagt wanneer we dit vergelijken met C-SPARQL and CQELS, en dit was een van onze belangrijkste doelen. We zien dat de client nu betaalt voor het grootste deel van de query uitvoering, wat veroorzaakt wordt door het gebruik van Triple Pattern Fragments. Het client CPU verbruik voor onze implementatie piekt in het begin van de



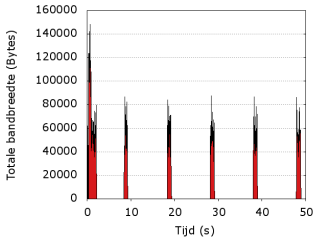
(A) De data overdracht in bytes gebruikmakende van **reification** voor een totale overdracht van 36.46 MB over 7924 bevragingen in dit tijdsinterval.



(B) De data overdracht in bytes gebruikmakende van **singleton properties** voor een totale overdracht van 2.57 MB over 627 bevragingen in dit tijdsinterval.



(C) De data overdracht in bytes gebruikmakende van **explicit graphs** voor een totale overdracht van 2.10 MB over 557 bevragingen in dit tijdsinterval.



(D) De data overdracht in bytes gebruikmakende van **implicit graphs** voor een totale overdracht van 6.70 MB over 1191 bevragingen in dit tijdsinterval.

FIG. 5: De data overdracht in bytes voor de vier methoden van annotatie voor een duur van 50 seconden. Deze plots gebruiken vervaltijden zonder caching. Reification gebruikt verbruikt de meeste bandbreedte, terwijl graphs het minst verbruikt.

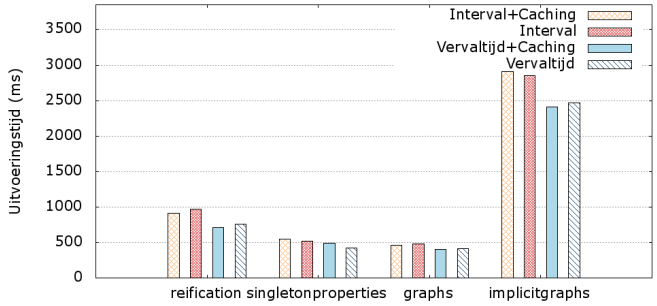


FIG. 6: Histogram van de uitvoeringstijden van de voorverwerkingen voor de verschillende opties van annotatie, gegroepeerd op methode van annotatie. De graph methode heeft de laagste uitvoeringstijden voor de voorverwerking. Vervaltijden zijn overal iets sneller en caching heeft geen significant invloed.

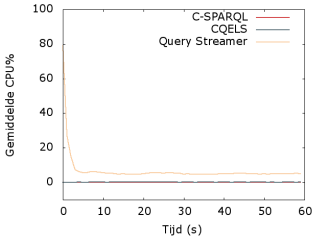
query intialisatie door de herschrijf fase, maar deze zakt daarna naar ongeveer 5%.

VII. CONCLUSIE

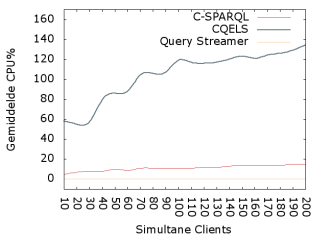
We hebben verchillende methoden voor continu actualiserende SPARQL bevragingen onderzocht en vergeleken, samen met een oplossing gebaseerd op Triple Pattern Fragments. Onze oplossing blijkt een significante reductie in server CPU verbruik te veroorzaken ten koste van een toename in bandbreedte en client verwerking.

We hebben vier verschillende methoden van annotatie onderzocht, waarvoor graphs de beste bleek te zijn in onze experimenten. De twee temporele domeinen, tijd intervallen en vervaltijden, kunnen gebruikt worden, waarvan de laatste de beste oplossing is in het geval van zeer volatiele data.

De oplossing die hier voorgesteld werd heeft enkel een extra laag bovenop de bestaande TPF client nodig om query streaming



(A) Het gemiddelde client CPU verbruik van één query voor C-SPARQL, CQELS en de oplossing die wij voorstellen. Initieel is het CPU verbruik voor onze implementatie zeer hoog, waarna deze convergeert naar ongeveer 5%. Het verbruik voor C-SPARQL en CQELS is vrijwel onbestaande.



(B) Het gemiddelde server CPU verbruik voor een toenemend aantal clients voor C-SPARQL, CQELS en de oplossing die wij voorstellen. Het CPU vebruik van onze oplossing blijkt veel minder beïnvloed te zijn door het aantal clients. Merk op dat de test machine vier core ter beschikking had.

FIG. 7: Het client en server CPU verbruik voor één query stream voor C-SPARQL, CQELS en de oplossing voorgesteld in dit werk.

toe te laten. De data die moet aanzien worden als dynamisch moet geannoteerd worden met tijd informatie in de dataset.

VIII. TOEKOMSTIG WERK

Dit onderzoek is slechts een eerste benadering tot continue bevraging op basis van Triple Pattern Fragments waarvan verschillende aspecten nog steeds verbeterd kunnen worden.

- In plaats van één statische en dynamische query als resultaat van de query herschrijving te hebben, kunnen *meerdere queries* gebruikt worden met elk een verschillende volatiliteit. Op deze manier kunnen complexere queries met verschillende dynamische triple patronen mogelijks efficiënter gecached en bevraagd worden.
- Graph annotatie blijkt de meest efficiënte te zijn uit onze experimenten, maar *implicit graphs* annotatie kan op verschillende vlakken nog steeds verbeterd worden om mogelijks nog efficiënter te worden.
- In dit werk zijn we er van uitgegaan dat de verander tijden van dynamische data steeds geweten was door de data voorziener. *Vervaltijden en tijd intervallen bepalen* kan zeer complex worden en vereist mogelijks geavanceerde patroonherkenning algoritmen op de data verander historie.
- Indien de TPF client op een efficiënte manier `FILTER` ondersteuning zou kunnen toevoegen, zouden vervaltijden en tijd intervallen veel efficiënter opgezocht kunnen worden. Dit kan mogelijks weer ten koste gaan van een hoger server CPU verbruik.
- Zoals besproken werd in een verwant onderzoek [26] zal *statische achtergrond data* mogelijks niet altijd dezelfde blijven. Dus de *Caching* module zou hier rekening mee moeten houden.
- De resultaten die hier gepresenteerd werden kunnen mogelijks veel verschillen bij andere use cases, dus vergelijkbare testen voor andere query types kunnen mogelijks interessante resultaten produceren. Bevragingen voor meer statische data kunnen bijvoorbeeld efficiënter worden.
- Indien een grotere test omgeving beschikbaar zou zijn, zouden de server en client performantie experimenten voor een groter aantal simultane clients uitvoerd kunnen worden om de eigenlijke limieten van de server voor deze benadering te bepalen.

REFERENTIES

[1] Tim Berners-Lee. Linked Data, July 2006.
URL `http://www.w3.org/DesignIssues/LinkedData.`

html

[2] Ruben Verborgh, Miel Vander Sande, Pieter Colpaert, Sam Coppens, Erik Mannens, and Rik Van de Walle. Web-Scale Querying through Linked Data Fragments. In *Proceedings of the 7th Workshop on Linked Data on the Web*. April 2014.
URL http://events.linkeddata.org/ldow2014/papers/ldow2014_paper_04.pdf

[3] Tim Berners-Lee, James Hendler, Ora Lassila, et al. The Semantic Web. *Scientific american*, 284(5):28–37, 2001.
URL http://isel2918929391.googlecode.com/svn-history/r347/trunk/RPC/Slides/p01_theSemanticWeb.pdf

[4] JJ Carol and G Klyne. Resource Description Framework: Concepts and Abstract Syntax. Recommendation, W3C, February 2014.
URL http://www.w3.org/TR/2014/REC-rdf11-concepts-20140225/

[5] Eric PrudHommeaux, Andy Seaborne, et al. SPARQL Query Language for RDF. *W3C recommendation*, 15, 2008.
URL http://www.w3.org/TR/rdf-sparql-query/

[6] Steve Harris, Andy Seaborne, and Eric Prud'hommeaux. SPARQL 1.1 Query Language. Recommendation, W3C, March 2013.
URL http://www.w3.org/TR/2013/REC-sparql11-query-20130321/

[7] Nuno Lopes, Antoine Zimmermann, Aidan Hogan, Gergely Lukácsy, Axel Polleres, Umberto Straccia, and Stefan Decker. RDF Needs Annotations. In *W3C Workshop on RDF Next Steps, Stanford, Palo Alto, CA, USA*. Citeseer, 2010.
URL http://www.w3.org/2009/12/rdf-ws/papers/ws09

[8] Donald Ballou, Richard Wang, Harold Pazer, and Giri Kumar.Tayi. Modeling Information Manufacturing Systems to Determine Information Product Quality. *Management Science*, 44(4):pp. 462–484, 1998. ISSN 00251909.
URL http://www.jstor.org/stable/2634609

[9] Frank Manola, Eric Miller, and Brian McBride. RDF 1.1 Primer. Working group note, W3C, June 2014.
URL http://www.w3.org/TR/2014/NOTE-rdf11-primer-20140624/#section-turtle-family

[10] C. Gutierrez, C.A Hurtado, and A Vaisman. Introducing Time into RDF. *Knowledge and Data Engineering, IEEE Transactions on*, 19(2):207–218, Feb 2007. ISSN 1041-4347. doi:10.1109/TKDE.2007.34.
URL http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4039284

[11] Claudio Gutierrez, Carlos Hurtado, and Alejandro Vaisman. Temporal RDF. In *The Semantic Web: Research and Applications*, pages 93–107. Springer, 2005.
URL http://link.springer.com/chapter/10.1007/11431053_7

[12] David Beckett, Tim Berners-Lee, Eric Prudhommeaux, and Gavin Carothers. Turtle-terse RDF triple language. Recommendation, W3C, February 2014.
URL http://www.w3.org/TR/2014/REC-turtle-20140225/

[13] Gavin Carothers. RDF 1.1 N-Quads: A line-based syntax for RDF datasets. Recommendation, W3C, February 2014.
URL http://www.w3.org/TR/2014/REC-n-quads-20140225/

[14] Vinh Nguyen, Olivier Bodenreider, and Amit Sheth. Don't Like RDF Reification? Making Statements About Statements Using Singleton Property. In *Proceedings of the 23rd International Conference on World Wide Web*, WWW '14, pages 759–770. ACM, New York, NY, USA, 2014. ISBN 978-1-4503-2744-2. doi:10.1145/2566486.2567973.
URL http://doi.acm.org/10.1145/2566486.2567973

[15] E. Della Valle, S. Ceri, F. van Harmelen, and D. Fensel. It's a Streaming World! Reasoning upon Rapidly Changing Information. *Intelligent Systems, IEEE*, 24(6):83–89, Nov 2009. ISSN 1541-1672. doi:10.1109/MIS.2009.125.
URL http://www.few.vu.nl/~frankh/postscript/IEEE-IS09.pdf

[16] Davide Barbieri, Daniele Braga, Stefano Ceri, Emanuele Della Valle, and Michael Grossniklaus. Stream Reasoning: Where We Got So Far. In *Proceedings of the NeFoRS2010 Workshop, co-located with ESWC2010*. 2010.
URL http://wasp.cs.vu.nl/larkc/nefors10/paper/nefors10_paper_0.pdf

[17] Arvind Arasu, Brian Babcock, Shivnath Babu, John Cieslewicz, Mayur Datar, Keith Ito, Rajeev Motwani, Utkarsh Srivastava, and Jennifer Widom. STREAM: The Stanford Data Stream Management System. *Book chapter*, 2004.
URL http://ilpubs.stanford.edu:8090/641/1/2004-20.pdf

[18] Davide Francesco Barbieri, Daniele Braga, Stefano Ceri, Emanuele Della Valle, and Michael Grossniklaus. Querying RDF Streams with C-SPARQL. *SIGMOD Rec.*, 39(1):20–26, September 2010. ISSN 0163-5808. doi:10.1145/1860702.1860705.
URL http://doi.acm.org/10.1145/1860702.1860705

[19] Davide Francesco Barbieri, Daniele Braga, Stefano Ceri, and Michael Grossniklaus. An Execution Environment for C-SPARQL Queries. In *Proceedings of the 13th International Conference on Extending Database Technology*, EDBT '10, pages 441–452. ACM, New York, NY, USA, 2010. ISBN 978-1-60558-945-9. doi:10.1145/1739041.1739095.
URL http://doi.acm.org/10.1145/1739041.1739095

[20] Danh Le-Phuoc, Minh Dao-Tran, Josiane Xavier Parreira, and Manfred Hauswirth. A Native and Adaptive Approach for Unified Processing of Linked Streams and Linked Data. In *The Semantic Web–ISWC 2011*, pages 370–388. Springer, 2011.
URL http://iswc2011.semanticweb.org/fileadmin/iswc/Papers/Research_Paper/05/70310368.pdf

[21] Carlos Buil-Aranda, Aidan Hogan, Jürgen Umbrich, and Pierre-Yves Vandenbussche. SPARQL Web-Querying Infrastructure: Ready for Action? In *The Semantic Web–ISWC 2013*, pages 277–293. Springer, 2013.
URL http://link.springer.com/chapter/10.1007/978-3-642-41338-4_18

[22] Joycent, Inc. Node.js.
URL http://nodejs.org/

[23] iMinds iLab.t. Virtual Wall: wired networks and applications.
URL http://ilabt.iminds.be/virtualwall

[24] Chen Levan. CQELS engine: Instructions on experimenting CQELS.
URL https://code.google.com/p/cqels/wiki/CQELS_engine

[25] StreamReasoning. Continuous SPARQL (C-SPARQL) Ready To Go Pack.
URL http://streamreasoning.org/download

[26] Soheila Dehghanzadeh, Daniele Dell'Aglio, Shen Gao, Emanuele Della Valle, Alessandra Mileo, and Abraham Bernstein. Approximate Continuous Query Answering Over Streams and Dynamic Linked Data Sets. In *15th International Conference on Web Engineering*. s.n., Switzerland, June 2015.
URL http://dx.doi.org/10.5167/uzh-110296

Contents

Source Code

The implementation presented in this work can be found at `https://github.com/rubensworks/TPFStreamingQueryExecutor` and the code for executing the experiments and plotting their results can be found at `https://github.com/rubensworks/TPFStreamingQueryExecutor-experiments`.

Acronyms

API Application Programming Interface. 17

CQELS Continuous Query Evaluation over Linked Stream. iii, xxx, xxxi, xxxvi, xxxvii, 42, 44, 46, 69, 71–75, 80, 81, 88, 89, 91, 92, 98

C-SPARQL Continuous SPARQL. xxix–xxxi, xxxvi, xxxvii, 34–44, 46, 69, 71–75, 80, 81, 88, 89, 91, 92, 98

DIS Data Integration System. xxix, 27

DSMS Data Stream Management System. 33, 35, 38, 39

EDBC event-driven backward chaining. 40

EP-SPARQL Event Processing SPARQL. xxxvi, 39–42, 44, 46

HTTP Hypertext Transfer Protocol. 1, 5, 6, 14, 17, 54, 74, 96

IRI Internationalized Resource Identifier. 2, 30, 44

JSON Javascript Object Notation. xxvii, 9, 31

JSON-LD JSON Linked Data. 31

LDF Linked Data Fragment. 16, 17, 92, 96

OWL Web Ontology Language. 10, 11

R_2O Relational-to-Ontology. 41

RDF Resource Description Framework. xxix, xxxv, 2, 5–13, 29, 30, 32–36, 38, 39, 41, 44–46, 48, 53, 54, 56

RDFS RDF Vocabulary Definition Language. 10, 11

S_2O Stream-to-Ontology. 41, 46

SPARQL SPARQL Protocol and RDF Query Language. xxix, xxxiii, xxxv–xxxviii, 2–6, 9, 12–16, 19, 21, 30, 33–36, 38–46, 54, 58, 60, 61, 63–67, 69, 75, 77–79, 81, 91, 92, 94

SPARQL$_{Stream}$ SPARQL Stream. xxix, xxxvi, 41–43, 46, 97

TA-RDF Time-Annotated RDF. 44

TA-SPARQL Time-Annotated SPARQL. xxxvi, 39, 44–46

TPF Triple Pattern Fragment. xxix, xxxiii, xxxv, 2, 16–19, 21, 22, 54, 56–58, 60, 62, 63, 65, 69, 71, 74, 77, 81, 91, 92, 94–97

URI Uniform Resource Identifier. xxxv, 5, 6, 8–10, 13, 19, 22, 36, 38, 49, 54, 95

VOID Vocabulary of Interlinked Datasets. 17

W3C World Wide Web Consortium. 30

XML EXtensible Markup Language. 9

List of Figures

List of Tables

Listings

Chapter 1

Preface

1.1 Introduction

Information is becoming more important each day. Smart clients keep us up-to-date with everything we need to know. A lot of this information transfer happens over HTTP, the base protocol of the *World Wide Web*. The web is a collection of interlinked documents aimed at providing information. The primary objective of the web is to provide information to people, which can be read with web browsers.

A lot of end-user applications require real-time data updates. For example, applications that provide train schedules might also serve info about the delay of those trains. This type of data can remain the same up until the final hours or minutes before the train arrives. The moments on which this data changes can also be estimated by the data provider when for example the train delay calculation is done once each minute. In this case, the data provider can inform the user that this data *could* change after one minute. The client could then refresh the data after that minute and pull new data if a newer version exists.

Application scenarios as this one do not require data updates within a couple of milliseconds after the data is changed. A user would not mind if the delay info about his train is three seconds behind on the actual data. Background processes in such applications could require less precise and less frequent updates. And more importantly, the client should be able to choose when to ask for new data updates. It is possible that the train the user has to take does not leave for another 24 hours, so there is no need for querying the delays of this train yet, or at least at a much lower frequency.

1.2 Problem Statement

Linked Data [12] is an instantiation of the Semantic Web [13] idea in which RDF and SPARQL are the standard technologies to respectively declare data in a triple structure and query that data. Machines should be able to independently discover information and reason about it.

RDF is a framework to declare relations between resources, these relations are represented as triples and are called *facts*. Each triple is built with three elements: *a*) The subject of the fact, which is mostly a resource that is represented as an IRI. *b*) The predicate that indicates the type of relation this fact represents between the subject and the object. *c*) The object of the relation, which is a resource or literal. Even though applications can search for information they need in a list of triples, it might be better to use the SPARQL query protocol to search in such a list. SPARQL endpoints allow clients to execute SPARQL queries. These queries are based on triple patterns, which means that parts of the triple can be made up of variables with the goal of finding matching values for the variables.

Traditional SPARQL endpoints have some performance issues because of the unbounded complexity of SPARQL queries. Triple Pattern Fragments (TPFs) [3] aim to solve this issue by partially moving this complexity to the clients, and thereby reducing the computational load on SPARQL endpoints.

Many approaches [7, 14, 9, 15, 10, 11] for querying Linked Data streams with a variable rate of data generation already exist. While traditional SPARQL querying and data representation is based on static data, most of these new approaches introduce ways of representing RDF triples in a time-sensitive way. They also introduce methods to query and reason over this data, combined with the traditional static data. Clients who wish to receive continuous updates on their time-sensitive queries simply need to register their query to a query endpoint and the server will take care of the rest.

The existing methods for querying time-sensitive information with SPARQL are prone to server overloading. The first cause of this scalability problem is the unbounded complexity of queries, which can be solved by using TPF. The second reason is that the query endpoints accept query registrations, which means that servers need to keep states of each registered query and must continuously process each of them instead of the client continuously polling for updates. As long as the number of queries is guaranteed to be limited, this approach should not cause many problems. But for public endpoints, this can become a major cause of server downtime.

The goal of this work is to find a way to make time-sensitive SPARQL querying more scalable. This will be done by extending the Triple Pattern Fragments approach while making sure that the clients should not register queries to the server and thus keeping the servers stateless.

1.3 Research Questions

The main research question that will be investigated in this work is:

- *How much can we improve the efficiency of polling-based SPARQL querying when knowing the temporal query results volatility?*

In other words, we will investigate different approaches on how to perform SPARQL queries on real-time data based on polling, with the assumption that the data provider has knowledge of the frequency the data changes. For answering this question, several other sub-questions will also need an answer:

1. *How can clients make use of this volatility knowledge for improving the efficiency of SPARQL querying?*
2. *Which volatility metadata can be added to the query results?*
3. *Which approaches can we distinguish for adding temporal volatility knowledge?*
4. *How does this use of volatility knowledge perform against existing SPARQL engines?*
5. *Which parameters are required to adequately measure this efficiency?*

The first sub-question requires clients to somehow be notified by the SPARQL endpoint of the volatility metadata. For sub-question 2, a comparison of different possible metadata has to be made. Sub-question 3 will require a certain data type to be used for sending this volatility metadata, together with the original data. Different approaches can be compared for this. The fourth sub-question requires an implementation of at least one of the approaches to be compared with the traditional alternatives. The last sub-question is needed to determine which parameters are required for doing this comparison.

Another sub-question that could have been added is: *How to predict the volatility of query results?* This is out of the scope of this work, since that could easily become a completely separate research on its own. However, this does not make this prediction unimportant, this is definitely something that has to be researched in future work.

1.4 Hypotheses

Following hypotheses related to the research questions have been identified.

1. *Some volatility patterns are too complex to efficiently predict and declare.*
2. *Well calculated data change patterns improve the efficiency of SPARQL querying.*

3. *Good usage of volatility knowledge in client polling can make SPARQL querying more efficient than existing systems in certain scenarios.*
4. *Smart client polling is more scalable for the server than streaming-based SPARQL engines.*

These will either be confirmed or denied at the end of this work.

1.5 Outline

The remainder of this document is structures as follows. Chapter 2 will give an overview of the relevant *Semantic Web* technologies. After that, a basic use case will be explained in Chapter 3 which will be used throughout this document. Chapter 4 will explain the terminology and concepts that will be used in the context of dynamic data. An overview of the relevant methods of *RDF annotation* will be given in Chapter 5. A selection of the most important *Streaming RDF* approaches will be presented in Chapter 6. Next, different methods for representing dynamic data are discussed in Chapter 7. In Chapter 8, the architecture of our proposed solution is explained. After that, our previously presented use case will be expanded in Chapter 9 for the different methods for representing dynamic data, together with an overview of the executed experiments of which the results are presented in Chapter 10. Finally, some concluding remarks and a discussion of possible future work will be discussed in respectively Chapters 11 and 12.

Chapter 2

Semantic Web

2.1 Semantic Web in general

The World Wide Web is a global system for interlinked documents. The idea behind this Web is to enable humans to retrieve data and draw conclusions. In 2001, Tim Berners-Lee [13] proposed an evolution towards a *Semantic Web* where the documents are not only readable by humans, but also interpretable by machines. The ultimate goal of this evolution is to reach *Semantic Interoperability*, so that humans and machines can derive the same *knowledge* from data.

Tim Berners-Lee also coined the term *Linked Data*, which he calls *"The Semantic Web done right."* [16]. While the Semantic Web is about machine-accessible data, for which Linked Data is the framework, using technologies like RDF and SPARQL, which will be explained later.

2.1.1 The Four Rules of Linked Data

Tim Berners-Lee outlined a set of rules [12, 13] for publishing data on the Web as *Linked Data* to become *semantically interoperable*:

- Use URIs as names for things.
- Use HTTP URIs so that people can look up those names.
- When someone looks up a URI, provide useful information, using the standards (RDF, SPARQL).
- Include links to other URIs so that they can discover more things.

The first rule simply states that any "thing", that can be referenced as data resource, should have a unique Uniform Resource Identifier. URIs are required because they are easily accessible to machines.

The second rule is also very straightforward, a URI is almost useless if it can not be looked up via the Hypertext Transfer Protocol protocol. The act of looking up data from a URI is called *dereferencing*, with the goal of discovering more information about a subject.

The third rule, that useful information should be provided, demands data providers to supply information using the standards for Linked Data representation: RDF and SPARQL. These standards will be explained in respectively Sections 2.2 and 2.3.

The last rule is very important for making an unbounded Web, just like the regular Web where everything is connected via hyperlinks.

2.1.2 Five-Star Rating

With the goal of encouraging data owners, Tim Berners-Lee [12] developed a five-star rating system that indicates how open their data is. This rating aims at making Linked *Open* Data, which is Linked Data, but released under an open license. This *open*ness is a very strong requirement for making a web of Linked Data, otherwise data might be publicly available without people being legally allowed to use it.

⋆	On the web, open license
⋆⋆	Machine-readable data
⋆⋆⋆	Non-proprietary format
⋆⋆⋆⋆	RDF standards
⋆⋆⋆⋆⋆	Linked RDF

Table 2.1: Five star rating for Linked Open Data.

2.1.3 Semantic Web Stack

The *Semantic Web Stack* is hierarchy of languages and technologies that are standardized to make the Semantic Web possible. Like any layered architecture, each layer uses services provided by the layers below and offers services to the layers above. The parts this research will focus on, is querying with SPARQL together with the RDF data interchange. An overview of this architecture can be seen in Figure 2.1.

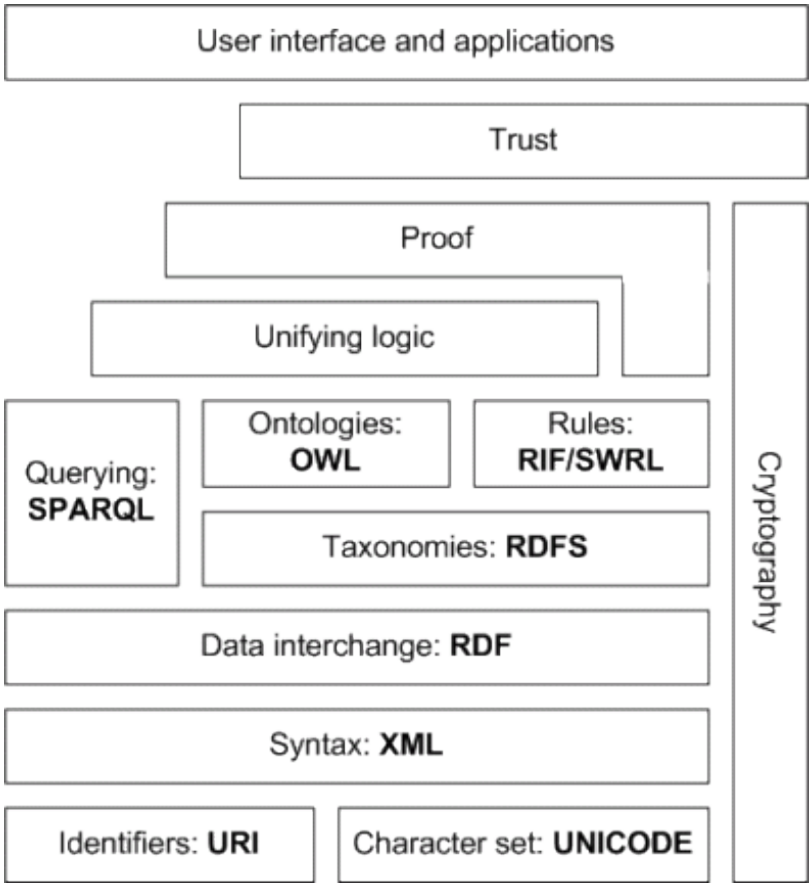


Figure 2.1: The Semantic Web Stack.

2.2 RDF

The Resource Description Framework (RDF) [17] provides a graph-based data model that can be used to structure data and interlink it. RDF uses so-called *triples* to represent data, these triples consist of a *subject*, *predicate* and *object*. A real-world resource (subject) can be related in some way (predicate) to another resource (object).

For example, the graph in Figure 2.2 can be serialized as the triples in Listing 2.1, this serialization format will be explained in Subsection 2.2.2. It is easy to understand that this data model is able to represent any form of data and its relationships.

Since RDF 1.1 [18] *graphs* are supported. This extends triples with a fourth element which can be used to add information about a triple. Before this, triple annotation had to be done with the use of *reification*. Triple annotation will be further explained in Chapter 5.

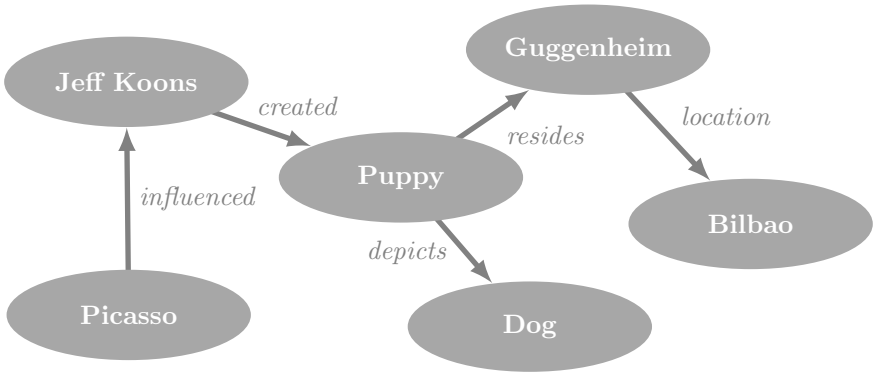


Figure 2.2: Example [1] of an RDF graph.

```
http://dbpedia.org/resource/Jeff_Koons http://dbpedia.org/ontology/created
     http://dbpedia.org/resource/Puppy .
http://dbpedia.org/resource/Picasso http://dbpedia.org/ontology/influenced
     http://dbpedia.org/resource/Jeff_Koons .
http://dbpedia.org/resource/Puppy http://dbpedia.org/ontology/resides http
    ://dbpedia.org/resource/Guggenheim .
http://dbpedia.org/resource/Puppy http://dbpedia.org/ontology/depicts http
    ://dbpedia.org/resource/Dog .
http://dbpedia.org/resource/Guggenheim http://dbpedia.org/ontology/
    location http://dbpedia.org/resource/Bilbao .
```

Listing 2.1: Example [1] of RDF triples.

2.2.1 Definitions

In what follows, graphs and triples will be formally defined based on previous work [19]. The notations $\mathcal{U}$, $\mathcal{B}$, $\mathcal{L}$ and $\mathcal{V}$ [20] will be used to respectively denote the sets of all URIs, blank nodes, literals and variables.

Definition 1 (RDF triples)**.** $\mathcal{T} = (\mathcal{U} \cup \mathcal{B}) \times \mathcal{U} \times (\mathcal{U} \cup \mathcal{B} \cup \mathcal{L})$ *is the (infinite) set of all* RDF triples*. One triple pattern can be represented as* $t = (s, p, o)$ *with* $s, p, o \in (\mathcal{V} \cup \mathcal{T})$*.*

URIs These are allowed for subjects, predicates and objects. They simply refer to a 'thing' that is defined as a separate resource. With the use of *dereferencing* [1, 21], clients can access specific entities by resolving their URI.

Blank nodes Blank nodes are anonymous nodes. For these nodes, no URI or literal is given because it is either not known or not important. These nodes can be labeled. Labeled blank nodes are assumed to refer to the same resource when using the same label. Unlabeled blank nodes can contain triple information about the resource.
In situations where a stronger identification is needed for these blank nodes, the technique called *skolemization* [18] can be used. This replaces blank nodes with a globally unique URI without changing their meaning.

Figure 2.3: Example of a triple where the subject is a blank node.

```
http://dbpedia.org/resource/Jeff_Koons http://xmlns.com/foaf/0.1/name "
    Jeff Koons" .
```

Listing 2.2: Triple where the object is a string literal.

Figure 2.3 shows an example of a triple where the subject is unknown and is represented as a blank node.

Literals Simple primitive types which can be strings, integers and any other datatype that can be referenced by a URI.

Listing 2.2

Variables Almost the same as labeled blank nodes, but with that difference that they are used for data extraction with for example SPARQL (more information in Section 2.3). These variables can not be used in data stores for declaring entities.

2.2.2 Serialization

RDF was at its launch in 2001 XML-based [22]. Because of the XML element, RDF/XML is inherently verbose [1] and introduces significant schema overhead compared to other technologies like JSON [23]. For this reason, the *Turtle* [24] syntax was introduced, for which an example was already shown in Listing 2.1. The remainder of this document will always use the Turtle syntax to represent RDF, unless specifically stated otherwise.

2.2.3 Turtle Syntax

Triples are represented by writing their elements (subject, predicate, object) separated by a whitespace and finally terminated by a dot. This syntax was constructed in such a way that it improves human readability by making it look like regular sentences. As stated in Definition 1, each of these elements can be part of different sets that each have their own syntax:

- $\mathcal{U}$: The URI surrounded by angle brackets. For example: `<http://example.org/York>`
- $\mathcal{B}$ (labeled): `_:` followed by the label of the blank node. For example: `_:york`

```
@prefix gh: <http://guggenheim.org/new-york/collections/collection-online/
    artwork/> .
@prefix dc: <http://purl.org/dc/terms/> .
@prefix viaf: <http://viaf.org/viaf/> .

gh:48 dc:creator viaf:5035739 .
```

Listing 2.3: Example [1] of the Turtle `@prefix` abbreviation for URIs.

```
gh:48 dc:creator viaf:5035739;
      dc:title "Puppy" .

viaf:5035739 :influencedBy viaf:15873,
                           viaf:95794725 .
```

Listing 2.4: Example [1] of the Turtle semicolon and comma abbreviation for shared subjects and/or predicates.

- $\mathcal{B}$ (unlabeled): Triple parts encapsulated between brackets. For example: `[] foaf:knows [ foaf:name "Bob" ] .`
- $\mathcal{L}$: Literal between double quotes. For example: `"Bob"`, `"42"^^xs:integer` or `"York"@en`

Some URIs can become quite long, because of that we can use a shorthand with the `@prefix` directive. An example of this mechanism can be found in Listing 2.3. This prefix makes our triples much shorter and allows prefixes to be reused across multiple triples. This reusability also reduces the chance on mistakes when otherwise duplicating these URIs.

Turtle also has built-in abbreviation possibilties for when triples have identical subjects or identical subjects and predicates. For this, semicolons and commas can respectively be used, an example can be seen in Listing 2.4.

Unfortunately, Turtle does not support graphs which were introduced in RDF 1.1 [18]. There are however some alternative ways of serializing RDF which are discussed in the RDF 1.1 primer [25]. For example, *TriG* [26] is an extension of Turtle which enables the specification of multiple graphs in one document. Any valid Turtle document is also a valid TriG document.

2.2.4 Vocabularies

Only the basics of RDF vocabularies (also known as *ontologies*) will be covered here, since these are not needed in very much detail for this research.

The RDF Vocabulary Definition Language (RDFS) [27] and the extension Web Ontology Language (OWL) [28] provide a foundation for creating vocabularies. These vocabularies can be used to define entities with their relations. Vocabularies are in fact

just triple collections in RDF defining classes and properties using the terms provided by RDFS and OWL.

Because of the open idea behind Linked Data, anyone can publish vocabularies and make use of them. This idea can also introduce some partially redundant vocabularies. This is why terms in vocabularies can also be linked to define their relation to each another.

Here are some basic examples [29] showing the power of RDFS:

- The entity `book:uri` is an element of the class `ex:Textbook`: `book:uri rdf: type ex:Textbook .`
- `book:uri` (entity) is also an element of class `ex:WorthReading`: `book:uri rdf: type ex:WorthReading .`
- Every *textbook* (class) is a *book* (class): `ex:TextBook rdfs:subClassOf ex:Book .`

2.2.5 Temporal RDF

Traditional RDF triples are not able to express the time and space in which facts are true. In domains where data needs to be represented for certain times or time ranges, these traditional representations should be altered. This section will discuss the introduction of time into RDF based on previous research [30, 31].

Definitions Two mechanisms for adding time were researched [31]: *versioning* and *time labeling*. 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. The reasons for this are that *a*) RDF is supposed to be extensible, and this labeling supports this, and *b*) complete snapshots every time a small part of the graph changes can introduce overheads.

Two types of temporal dimensions are considered [31]: *valid* and *transaction* times. Valid time is the time at which the element is valid in the world. The transaction time is the time instant at which the data was stored or created.

A distinction [31] is made between *point-based* and *interval-based* labeling. The former states info about an element at a certain time instant, while the former states info at all possible times between two time instants. It is also noted that these representations are interchangeable [30].

Vocabulary A *temporal vocabulary* was introduced [30] for mechanisms previously discussed, this will be referred to as `tmp` in the remainder of this document. The following properties are introduced:

```
_:stmt rdf:subject :me ;
       rdf:predicate foaf:workplaceHomepage ;
       rdf:object <http://me.example.org/> ;
       tmp:interval [ tmp:initial
                      "2008-04-01T09:00:00Z"^^xsd:dateTime ;
                      tmp:final
                      "2009-11-11T17:00:00Z"^^xsd:dateTime ] .
```

Listing 2.5: A time-annotated triple using reification representing the valid time in an interval-based representation.

```
_:stmt rdf:subject :me ;
       rdf:predicate foaf:workplaceHomepage ;
       rdf:object <http://me.example.org/> ;
       tmp:instant [ "2008-04-01T09:00:00Z"^^xsd:dateTime ,
                     "2008-04-01T09:00:01Z"^^xsd:dateTime ,
                     "2008-04-01T09:00:02Z"^^xsd:dateTime ,
                     ...
                     "2008-10-11T16:59:58Z"^^xsd:dateTime ,
                     "2008-10-11T16:59:59Z"^^xsd:dateTime ,
                     "2009-11-11T17:00:00Z"^^xsd:dateTime ] .
```

Listing 2.6: A time-annotated triple using reification representing the valid time in an point-based representation.

- `interval`: Used to indicate that the object element refers to an interval-based time representation, the referred object must contain the properties `initial` and `final`.
- `instant`: Used to indicate that the object element refers to an point-based time representation.
- `initial`, `final`: Indicates that the object element refers to respectively the start and the end of the interval-based time representation.

The range of `instant`, `initial`, `final` is the set of natural numbers representing timestamps or the `xsd:timestamp` data type. The literal `now` is introduced to represent the current time.

Example An example of the valid time of a triple in an interval-based representation can be found in Listing 2.5, this representation can be transformed to a point-based representation by listing all possible time instants between the interval edges at the cost of more triples, as can be seen in Listing 2.6.

2.3 SPARQL

RDF triple stores can be queried using the SPARQL Protocol and RDF Query Language (SPARQL) [32, 2]. A *SPARQL endpoint* is a SPARQL query engine which can be used

```
PREFIX dbpedia: <http://dbpedia.org/resource/>
PREFIX dbpprop: <http://dbpedia.org/property/>

SELECT ?work WHERE {
    ?work dbpprop:artist dbpedia:Pablo_Picasso .
}
```

Listing 2.7: A SPARQL query [1] to lookup all the works of `Picasso`.

to execute such queries targeted at certain datasets. The query language is based on pattern matching in RDF graph structures by using so-called *graph patterns*. As mentioned in Subsection 2.2.1, variables can be used instead of the subjects, predicates or objects with the goal of finding matches for these variables.

The example in Listing 2.7 has a defined predicate and object, but a variable subject. This means that all possible triples which have the same predicate and object will be collected, and in this case also selected because the variable is also available in the `SELECT` statement.

The graph pattern declared after the `WHERE` statement can be an arbitrarily complex RDF structure which has to be matched. It allows any number of tuples, including URIs, (nested) blank nodes and literals.

The SPARQL specification [32] allows for four different variations:

- `SELECT`: Extract raw values from an endpoint for variables which are returned in a table format.
- `CONSTRUCT`: Same as `SELECT`, but returns the data in valid RDF.
- `ASK`: Ask for `True` or `False`.
- `DESCRIBE`: (Partially) extract an RDF graph from an endpoint, the exact part of the graph depends on the endpoint.

2.3.1 Graphs

Selecting the RDF base graph can be done with the `FROM` clause which takes a URI as parameter. A so-called *named graph* can be selected with `FROM NAMED`. Named graphs can be used in the query statement for sub-graph scoping.

Sub-graphs can be used for scoping `FILTER` clauses, declaring `OPTIONAL` parts or taking a `UNION`s of them. More details on this can be found in the SPARQL specifications [32].

An example of a nested graph can be found in Listing 2.8

```
SELECT ?name ?mbox ?hpage
FROM <http://example.org/dft.ttl>
FROM NAMED <http://example.org/>
WHERE {
  {
      ?x foaf:name ?name .
      ?x a ex:Person .
      GRAPH <http://example.org/> {
          ?x foaf:mbox ?mbox
          OPTIONAL { ?x foaf:homepage ?hpage }
      }
  } UNION {
      ?x foaf:name ?name .
      ?x a ex:Company .
  }
}
```

Listing 2.8: Complex graphs in SPARQL.

```
PREFIX  <http://books.example/>
SELECT SUM(?lprice) AS ?totalPrice
WHERE {
  ?org :affiliates ?auth .
  ?auth :writesBook ?book .
  ?book :price ?lprice .
}
GROUP BY ?org
HAVING (SUM(?lprice) > 10)
```

Listing 2.9: Aggregate functions in SPARQL [2].

2.3.2 Aggregate functions

As of SPARQL 1.1 [2], aggregates were introduced. This opened up a whole new world of possible queries. For example: What is the total price of all the items that can be bought in shop x? The `SELECT` statement allows expressions to be enclosed into these aggregate functions to make them return one numerical value, instead of one or more values from triples. The allowed functions are: `COUNT`, `SUM`, `AVG`, `MIN` and `MAX`.

The example shown in Listing 2.9 will look for all the books from organizations having a total book price larger than 10, and it will return the total sum of those prices.

2.3.3 SPARQL endpoints

Triple stores can provide a SPARQL interface over HTTP [33], like Virtuoso [34] and Jena TDB [35], turning them into a SPARQL endpoint.

SPARQL endpoints act as simple web service. These endpoints implement the SPARQL protocol [32] by accepting SPARQL-queries. These queries can then be processed by the server and the results are then sent back to the client.

Since SPARQL-queries have no limitation on their range and complexity, the servers use a significant portion of processor time and memory for certain queries. This requires SPARQL endpoints to be high-performant and have a high-availability, since they should always be able to provide data. Unfortunately, these properties make endpoints poorly scalable. This issue is handled further in Section 2.4.

2.3.4 Definitions

Continuing on the definition of triples in Subsection 2.2.1 and previous work [20, 7], a collection of definitions will be presented to formally define the features of SPARQL.

Mapping A *mapping* μ is defined as a partial function $\mu : \mathcal{V} \rightarrow \mathcal{T}$, this represents the computation of bindings for the variables of a SPARQL query. $dom(\mu)$ is the subset of variables $\mathcal{V}$ where μ is defined, and $deg(\mu)$ is the cardinality of $dom(\mu)$.

Triple Pattern A *triple pattern* can be represented as $t = (s, p, o)$ with $s, p, o \in (\mathcal{V} \cup \mathcal{T})$.

Graph Pattern A *graph pattern* $\mathcal{P}$ is defined as a list of triple patterns t.

Compatible mappings Two given mappings μ_1 and μ_2 are *compatible* if $\forall x \in dom(\mu_1) \cap dom(\mu_2)$, then $\mu_1(x) = \mu_2(x)$.

Mapping operations Two mappings A and B can be combined in the following ways:
$A \bowtie B = \{\mu_1 \cup \mu_2 | \mu_1 \in A, \mu_2 \in B \text{ are compatible}\}$
$A \cup B = \{\mu | \mu \in A \text{ or } \mu \in B\}$
$A \setminus B = \{\mu \in A | \forall \mu_1 in B, \mu_1 \text{ and } \mu_1 \text{ are not compatible}\}$
$A \ltimes B = (A \bowtie B) \cup (A \setminus B)$ *The left outer-join operator*

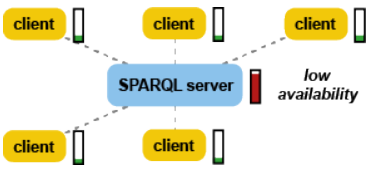
Graph Pattern Evaluation The notation $[[\mathcal{P}]]_{\mathcal{D}}$ is used to represent the evaluation of a graph pattern $\mathcal{P}$ over a dataset $\mathcal{D}$. It is recursively defined as:
$[[t]]_{\mathcal{D}} = \{\mu | dom(\mu) = var(t) \wedge \mu(t) \in \mathcal{D}\}$, with t a triple pattern and $var(t)$ the set of variables that occur in t.
$[[(\mathcal{P}_1 \texttt{ AND } \mathcal{P}_2)]]_{\mathcal{D}} = [[\mathcal{P}_1]]_{\mathcal{D}} \bowtie [[\mathcal{P}_1]]_{\mathcal{D}}$
$[[(\mathcal{P}_1 \texttt{ OPTIONAL } \mathcal{P}_2)]]_{\mathcal{D}} = [[\mathcal{P}_1]]_{\mathcal{D}} \ltimes [[\mathcal{P}_1]]_{\mathcal{D}}$
$[[(\mathcal{P}_1 \texttt{ UNION } \mathcal{P}_2)]]_{\mathcal{D}} = [[\mathcal{P}_1]]_{\mathcal{D}} \cup [[\mathcal{P}_1]]_{\mathcal{D}}$

2.4 Triple Pattern Fragments

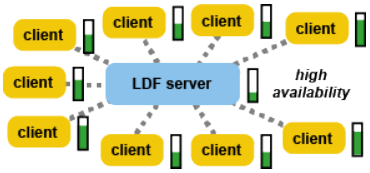
There is a major problem [3] with the performance and availability of existing SPARQL endpoints. Experiments [36] made it clear that only 30% of the public endpoints reach a monthly 'two nines'[1] of uptime, which is very low for a web service. The cause of this are the conflicting requirements of unrestricted SPARQL queries and public availability to many simultaneous users. Currently, clients can simply send queries of an arbitrary complexity. These complex queries introduce a bottleneck, because they block other pending queries. And when too many complex requests are sent, the server will simply go down under the high load.

2.4.1 Proposed Solution

Triple Pattern Fragments (TPFs) [3] are introduced as a solution to this issue. The idea is to put more computational responsibility at the clients and thereby reducing the computational load on the servers at the cost of increased data transfer. Figure 2.4 (taken from [3]) shows an overview of this idea, the bars next to the clients and servers indicate the relative need of processing power and the width of the dotted lines indicate the required data transfer.



(a) Classic scenario of high server load and clients waiting for query results.



(b) Lowering TPF server load by moving query execution to clients.

Figure 2.4: Comparison of the classic SPARQL endpoints and the TPF concept.

This solution causes the (simple) query results from servers to be rather large because queries are less detailed and large parts of the data will have to be processed client-side.

2.4.2 Definitions

The terms *Linked Data Fragment* and *Triple Pattern Fragment* (formerly know as *basic Linked Data Fragment*) are defined here based on definitions from previous work [3, 19].

Definition 2 (Linked Data Fragment (LDF))**.** *A Linked Data Fragment [38] of a Linked Data dataset is a resource consisting of those elements of this dataset that match a specific selector, together with their metadata and the controls to retrieve related Linked Data Fragments.*

[1] The 'amount of nines' [37] is derived from a percentage that indicates the uptime, or availability, of a system. The number of nines is calculated as $-log(1 - A)$, with A the percentage of availability.

Definition 3 (Triple Pattern Fragment (TPF))**.** *A Triple Pattern Fragment [39] is a Linked Data Fragment with a triple pattern as selector, count metadata, and the controls to retrieve any other triple pattern fragment of the same dataset, in particular other fragments the matching elements belong to.*

2.4.3 Metadata

Because of the large amount of data transfered per query, extra *metadata* and *controls* are sent together with the results. If for example every possible person is being queried, the result would be too big to send in one message. That is why *pagination* [40] is introduced together with the controls to *browse* through these pages. These controls are represented using the Hydra hypermedia API vocabulary [41].

These Hydra controls are not limited to pagination, but the Hydra Core Vocabulary (which is implemented by TPFs) allows for generic controls to be declared so that clients can autonomously discover how to use an API. A dataset description is also added in TPFs using Vocabulary of Interlinked Datasets (VOID) [42] for representing info such as the amount of triples. The importance of this triple count will be explained in Subsection 2.4.5.

2.4.4 Server

A TPF server [3] is a server that provides at least one dataset in a triple-based representation. The TPF server was designed with the goal of keeping response times in the same range of the average HTTP server, a few hundred milliseconds, while also being scalable. The most important requirement was to make sure the server could always be reached by the clients.

To comply with the defined requirements, two key decisions are made with regards to the server architecture. The first decision was that the resources offered by the server can not take too much processing time to generate. Second, partitioning of data into TPFs should be done in such a way that it can be reused for efficient caching. An example of such an implementation can be found on `http://linkeddatafragments.org/software/`.

The TPF server implementation can be tested at `http://data.linkeddatafragments.org/`. When requesting this page with the `Accept: text/turtle` header, a TPF response is given with the required controls discover the API and to perform additional queries, just like the visual forms a human would see when visiting that url. An example of these controls for one available dataset can be found in Listing 2.10. The executed query to get this full result was: `curl -H "Accept: text/turtle" http://data.linkeddatafragments.org/dbpedia`

```
@prefix hydra: <http://www.w3.org/ns/hydra/core#>.
@prefix void: <http://rdfs.org/ns/void#>.
@prefix : <http://data.linkeddatafragments.org/>.

...

<http://data.linkeddatafragments.org/dbpedia#dataset> a void:Dataset,
    hydra:Collection;
    void:subset :dbpedia;
    void:uriLookupEndpoint "http://data.linkeddatafragments.org/dbpedia{?
        subject,predicate,object}";
    hydra:search [
        hydra:template "http://data.linkeddatafragments.org/dbpedia{?
            subject,predicate,object}";
    ] .
    hydra:mapping [hydra:variable "subject";   hydra:property rdf:subject
        ],
                  [hydra:variable "predicate"; hydra:property rdf:
                      predicate],
                  [hydra:variable "object";    hydra:property rdf:object]
                      .
:dbpedia a hydra:Collection, hydra:PagedCollection;
         dcterms:title       "A 'dbpedia' Linked Data Fragment"@en;
         dcterms:description "Triple Pattern Fragment of the 'dbpedia'
             dataset containing triples matching the pattern { ?s ?p ?o
             }."@en;
         dcterms:source      <http://data.linkeddatafragments.org/dbpedia#
             dataset>;
         hydra:totalItems    "427670470"^^xsd:integer;
         void:triples        "427670470"^^xsd:integer;
         hydra:itemsPerPage  "100"^^xsd:integer;
         hydra:firstPage     <http://data.linkeddatafragments.org/dbpedia?
             subject=&predicate=&object=&page=1>;
         hydra:nextPage      <http://data.linkeddatafragments.org/dbpedia?
             subject=&predicate=&object=&page=2> .

...
```

Listing 2.10: TPF controls in the Turtle representation.

```
1 PREFIX dbpedia: <http://dbpedia.org/ontology/>
2 PREFIX foaf: <http://xmlns.com/foaf/0.1/>
3
4 SELECT ?p ?c
5 WHERE {
6     ?p a dbpedia:Artists .
7     ?p dbpedia:birthplace ?c .
8     ?c foaf:name "York"@en .
9 }
```

Listing 2.11: SPARQL query [3] that will be executed by the TPF client.

2.4.5 Client

The TPF client was designed in such a way that it will always try to optimally calculate query results in terms of computational usage. As explained before, a query will be calculated at the client, and only parts of it will be requested from the server.

If the query from Listing 2.11 would be executed, then the client would split this up into three parts. These parts are exactly the lines 5, 6 and 7, which will now be referred to as $F1$, $F2$ and $F3$. The main idea of the client algorithm [3] is that it will always join the fragments (in this case $F1$, $F2$ and $F3$) in such a way that recursively always the smallest fragment is taken first. This is where the triple count property as mentioned before comes in. Because the TPF server always sends an estimated triple count, the client will always know the smallest fragment to start from.

If for example $F1$ would be the smallest fragment with only 12 matches, this fragment would be the starting point. An iterator will run over these 12 matches and recursively apply the same algorithm but in this case with only the fragments $F2$ and $F3$ with the bound variable(s) matched from $F1$.

Eventually, all these recursive invocations would make up a complete result for the given query. It is clear that the reduced server load requires more requests. Instead of sending one complex SPARQL query to the server with the client waiting for a result, the client now calculates the query itself while requesting all the required fragments from the required URIs.

2.4.6 Results

From experiments performed [3] it is clear that the Triple Pattern Fragments approach effectively increases the average availability. However, there are some drawbacks. The article shows that traditional SPARQL querying is significantly more performant (as long as the endpoints are available), this is no surprise when taking into account the increased number of request. Another drawback is mentioned, which says that clients where bandwidth is limited (mobile phones), this increase request count could be a problem. This is however not such a big problem since TPF allows for better query caching.

Chapter 3

Use Case: Basic Data Model

This chapter briefly discusses a use case regarding SPARQL queries on train departure information. This use case will be used in the remainder of this document to illustrate certain concepts. This chapter introduces a *basic data model* which will be extended in Chapter 9 to perform measurements of the solution proposed in this work.

3.1 Problem Statement

We want to be able to request information about all train departures in the current train station. Figure 3.1 displays the *basic data model* used to represent data departures. We refer to this as the basic model, since later on different derived data models will be used in the context of time annotation. A custom train ontology will be used to represent these attributes as triples and will be referred to using the prefix `t:`. Table 3.1 shows a description list of the information associated with a train departure. This use case will assume a separate endpoint for each train station, so the train station itself will not be part of the query.

The data will always contain an instance of this data model for each train that is scheduled to depart from this station. When a new train is scheduled, a new instance will be added. Each data model instance has three attributes that will remain the same for its complete lifetime, these are marked in light grey on Figure 3.1. It also has two attributes that can change in its lifetime, these are the nodes marked in dark grey on Figure 3.1. When a train has left the station, the data provider can remove these departure facts.

The data for this use case is retrieved from the iRail API [43] which is able to fetch train departure information for all train stations in Belgium. Only the departures from station "Gent-Sint-Pieters" are used, to simulate a user retrieving information for this station. A Triple Pattern Fragments server will be used to hold this information. Every ten seconds a request for all coming train departures at "Gent-Sint-Pieters" will be done

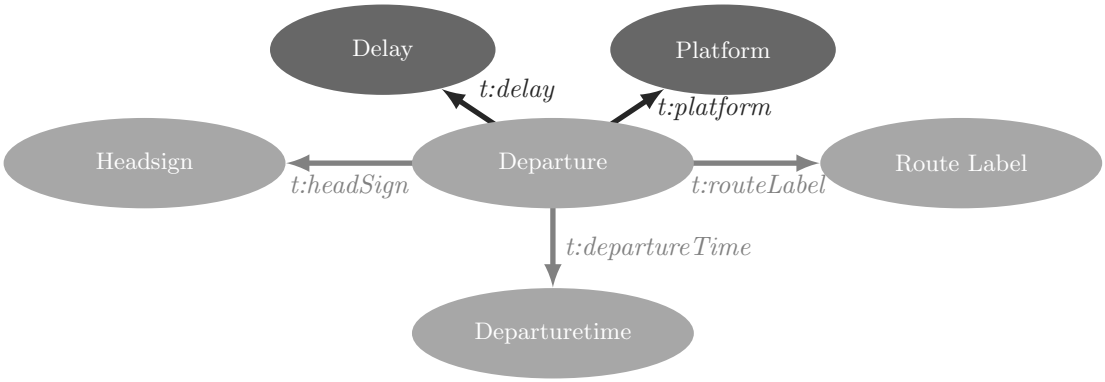


Figure 3.1: Basic data model for representing train departures in one train station. The dark grey nodes refer to dynamic data while other nodes are static.

Attribute name	**Description**	**Datatype**
Headsign	Destination of the train as shown to the passengers.	xsd:string
Departure time	Scheduled departure time of the train in this station.	xsd:dateTime
Route label	Type of route followed by the train.	xsd:string
Platform	Station platform this train will depart from.	xsd:string
Delay	Train delay compared to its original departure time.	xsd:duration

Table 3.1: Description of all train departure attributes used in the basic data model.

using the iRail API. Each departure already has a unique URI, so this is reused as the resource around which all attributes about this departure will be added as triples. If a certain departure is not yet available on the TPF server, a new instance from the basic data model will be created as triples using the information from the API. The dynamic triples about the delay and platform will receive a time annotation in a certain temporal domain. For each subsequent request, the dynamic triples will be either updated or re-created with another temporal range if their value remains the same, or they will be (partially) re-created when their value changes. The chosen type of time annotation and temporal domain will have a strong influence on this dynamic attribute update behavior, so each of them will be explained in Chapter 7.

Chapter 4

Data Freshness

Simply stated, data freshness is a concept that can measure how old certain data is. This can answer questions like: When was this data originally created/measured? Is my current data version up-to-date with the known truth? Is this data fresh enough to answer my questions within an accepted error margin?

This work aims on having a decent freshness of query results, so a review of data freshness in general is in order. This overview is largely based on a framework for the analysis of data freshness presented in previous work [44].

Data freshness can be analyzed as a family of quality factors [45] which can be measured using corresponding metrics. Three sub-dimensions of freshness can be observed [44, 46]. The first type, also the most commonly used interpretation, is called *currency* [47]. It describes the *staleness* of data when compared to the source(s). The next type is called *volatility* [46], it indicates how long an item remains valid. The third type is called *timeliness* [48]. It is used to describe how old data is relative to the change frequency. In the following sections, these three interpretations will be further explained.

Even though the following explanation takes into account the need of data availability at the client, in some cases clients could only need data at certain times. This complicates the data freshness analysis, because the freshness measurements should only occur at the times the data is needed at the client. For example, weather info on a mobile phone is in some cases only required when the user looks at the screen, so as long as the data is available when the user looks at the screen, the data is "fresh".

4.1 Classification

4.1.1 Currency

Currency [47], also known as *staleness*, is a factor that shows the "gap" in time between the last data measurement or extraction from a certain source and the moment it is reflected at the client. It deals with the speed of data updates.

For example, certain display websites of stock markets display a live time-counter which shows the passed time starting from the moment the last update has been received. This time-counter shows the *currency* of the stock data with the gap in this case being between the time the central stocks server does a measurement and the time until the moment the user is looking at the new data (this moment is constantly moving forwards in time in this case). An important note in this example: In order to fully comply with the definition of *currency*, these counters should start their counting from the moment the server obtained the stock data instead of the moment the client receives the data. This is because in certain scenarios, like this example, the propagation time could be a signification contributor to the *staleness* measurements.

In fact, one could argue that the *currency* of this last example only gaps the time between the stock server data extraction moment and the moment the data is first displayed on the client machine. In this respect, the client is the machine instead of the user as in the first example. An important conclusion from this is that the start- and endpoint must always be well defined when using a *currency* metric.

4.1.2 Volatility

Volatility [46] indicates the frequency with which data can vary in time. Static facts like George Washington was the first president of the United States always remain true, so this will have a volatility of 0. An example of very volatile data are the oil prices, because the time a certain price remains valid is very short.

4.1.3 Timeliness

This factor captures the notion of change frequency or create frequency of data in a source.

For example, *timeliness* [48] indicates how frequently a simple cache is configured to update all its entries. Or how often send-alive updates are sent from sensors to a central system.

4.2 Measurement

The following metrics can be used to measure a quality factor of data freshness. Each of these metrics are also classified to one of the sub-dimensions explained in Section 4.1.

Currency metric (*currency factor*) This is the most logical derivation of a metric for *currency*. This metric measures the passed time between the data change time at the source and the delivery time at the client. Since change time of data is not always possible to exactly know, this can be estimated as the *data extraction* time at the source, so this could be for example a creation or update time of a tuple in a database. This estimation is also used in the following metrics. Caching systems also define this metric as *recency* or *age* and displays the time passed since a cache object was last cached. The time unit of this metric is not defined by this definition, anything that can represent time is allowed, for example milliseconds or years.

$$\text{Formula: } C = \textit{query-time} - \textit{extraction-time}$$

Obsolescence metric (*currency factor*) A discrete metric measuring the amount of updates (full or partial) to the source data since the moment the client last extracted data. This measurement can be done using an integrated audit tool that detects changes [49] or by post-processing log files. The *obsolescence* has a direct relation to the *change frequency* of data, so the one can be derived from the other. This metric is also often called the *age* in caching systems.

$$\text{Formula: } O = \textit{count}(\textit{data-updates-since-last-query})$$

Freshness-rate metric (*currency factor*) The freshness-rate is a metric representing the percentage of data (the level of data can be chosen, for example triples) that is up-to-date with the source data. Caching systems mostly refer to this metric simply as *freshness*. This ratio is usually expressed as a percentage.

$$\text{Formula: } F = \textit{count}(\textit{up-to-date-data})/\textit{count}(\textit{data})$$

Volatility metric (*volatility factor*) The volatility metric represents the inverse duration for which data remains valid.

$$\text{Formula: } V = 1/\textit{data-valid-duration}$$

Timeliness metric (*timeliness factor*) This is a direct metric derivation of the *timeliness* of data. It is measured as the time elapsed since the last data update in the source. It tells how appropriate the age of the data is for the given scenario. It is similar to *data volatility* which measures the time interval wherein data is valid.

Formula: $T = \textit{query-time} - \textit{data-update-time}$

When using the *currency metric* and *timeliness metric*, respectively the *extraction-time* and *data-update-time* can differ when having one or more data elements. We define that we always take the time of the oldest data element. When for example the *data-update-time* for two web pages is measured with an "Index" page that has been updated 10 seconds ago, and a "Contact" page which has been updated 60 seconds ago, then the collective *data-update-time* is 60 seconds, the age of the oldest page.

4.3 Dimensions for Analysis

Data freshness can be influenced by several different dimensions, the most important ones are discussed below.

4.3.1 Nature of Data

When analyzing change frequencies of data, an easy classification into three categories can be derived [44].

- *Stable data:* This is data that is assumed to be static. For example, names and birthdays of people. Even though this data is assumed to be static, some scenario's could allow this data to change anyways when for example an error in the data existed, even though this occurs rarely.
- *Long-term changing data:* Data that is not static, but does not change very often. For example, the current Pope does not change frequently. The "low frequency" of change is domain dependent.
- *Frequently changing data:* Data that is very volatile, this again is domain dependent. For example, the current temperature in Austin, Texas.

Frequently-changing data is often provided in a push-based streaming fashion, where clients register to a server that will immediately notify its listeners when a change occurs. When using polling, the clients have to continuously send requests to the server. In cases where the time that data remains unchanged can be predicted and modeled, the clients can much more efficiently send requests by only requesting at times the data is changed.

4.3.2 Application Types

Not only does freshness of data depend on the freshness of the *extracted* data, it also depends on the processes that are responsible for this extraction, together with the

integration and delivery of the data. The reason for adding a separate dimension, is the fact that each type of these processes can introduce some additional delay.

These types are observed in the context of so called Data Integration Systems (DISs). These systems are responsible for obtaining data from *data sources* (in the context of this research, also the server(s)), and passing this data to the client. This is illustrated in Figure 4.1

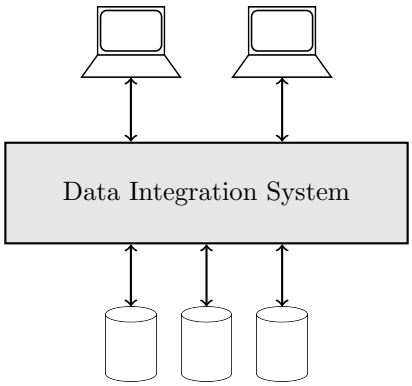


Figure 4.1: Data Integration System with data sources at the bottom and clients at the top.

- *Virtual systems:* No materialization of data, incoming queries are always fully calculated.
- *Caching systems:* Some or all data is cached. Estimations are done for how long data in the cache will be valid, when that time is passed, relevant parts of cache are invalidated so that the data in question has to be re-calculated.
- *Materialized systems:* Data is materialized at certain times and incoming queries are answered using parts of that previously materialized data.

Certain freshness metrics can only be applied in certain dimensions, Table 4.1 shows the co-relation of these dimensions.

	Frequently changing	**Long-term changing**	**Stable**
Virtual	Currency	Timeliness, Volatility	Timeliness, Volatility
Caching	Currency, Obsolescence, Freshness-rate	Timeliness, Volatility	Timeliness, Volatility
Materialized	Currency, Obsolescence	Timeliness, Volatility	Timeliness, Volatility

Table 4.1: Correlation of the data freshness dimensions in terms of the freshness metrics. Nature of Data in the columns and Application Types in the rows.

Chapter 5

RDF Annotations

In some cases, it is required to add additional info about certain data, this extra information is called metadata. We can for example have triples, which is our data, and want to annotate them with a certain expiration time, which is the metadata.

The concept of annotating RDF is discussed in previous work [4]. The researchers presented some domains, like temporal data, in which the annotation of triples is required. They compare reification with other *quad-centric* approaches by using a quad-supporting syntax like *N-Quads* [5]. Note that not all RDF serialization types support quads, so these *quad-centric* approaches can only be used in certain syntaxes. Another approach was presented [6] in which *Singleton Properties* are introduced for instantiating generic predicates and thereby allowing metadata to be added to relations.

5.1 Reification

The concept of *reification* allows you to use *subject*, *predicate* and *object* as predicates, which in combination with blank nodes allows the addition of annotations to RDF triples. An example of reification can be seen in Listing 5.1. The reason reification exists is because RDF did not include the concept of *graphs* until version 1.1, so another way of annotating triples was required.

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .

_:stmt rdf:subject :me ;
       rdf:predicate foaf:workplaceHomepage ;
       rdf:object <http://me.example.org/> .
```

Listing 5.1: The reified version of the triple `:me foaf:workplaceHomepage <http://me.example.org/> .`

```
SELECT ?name ?mbox ?hpage
WHERE {
  GRAPH ?g {
      ?x foaf:mbox ?mbox .
  }
  ?g a :certainDataset .
}
```

Listing 5.2: A SPARQL using the `GRAPH` keyword.

5.2 Singleton Properties

Researchers introduced *Singleton Properties* [6] as an alternative to reification. Which is, in contrast with graph-based approaches like N-Quads, fully compatibly with the original RDF 1.0. This means that this approach is fully triple-based. The idea is to create unique instances (singletons) of properties, which then can be used for further specifying that relationship by for example adding annotations. New instances of properties can be created by using the `sp:singletonPropertyOf` property.

If we would for example have a triple `:me foaf:workplaceHomepage <http://me.example.org/>`, then an instance of this relation can be created by writing the triple `foaf:workplaceHomepage#1 sp:singletonPropertyOf foaf:workplaceHomepage`, and the triple `:me foaf:workplaceHomepage#1 <http://me.example.org/>` would have exactly the same meaning. Note that the naming of these instances would be better if they were blank nodes, but since RDF does not allow blank nodes for properties, IRIs have to be used here. Annotations can be added to this new singleton property without affecting the parent property.

It is also noted [6] that creating singleton properties for each relation could introduce a lot of overhead, so this is not advised. These instances should only be created if annotations are required and in some cases it is advised to cluster singleton properties for relations with similar or equal metadata.

5.3 RDF Graphs

SPARQL standardized *named graphs* before RDF 1.1. It allows queries to contain scoped triple patterns, with that scope being a certain graph. That graph is just another resource that can be used inside other triples. An example for using the `GRAPH` keyword can be found in Listing 5.2.

N-Quads recently became a W3C recommendation, and is a variant of Turtle which allows encoding of multiple graphs. A paper [4] mentions two reasons for not using these quad-centric approaches: *a*) They say none of them are W3C recommendations, this is however not the case anymore, since february 2014 N-Quads has become a recommendation [5]. *b*) A lot of the existing RDF are not forwards compatible with this new syntax, this is still the case. N-Quads can be used for serializing triples with their

```
_:stmt rdf:subject :me ;
       rdf:predicate foaf:workplaceHomepage ;
       rdf:object <http://me.example.org/> ;
       tmp:interval [ tmp:initial
                      "2008-04-01T09:00:00Z"^^xsd:dateTime ;
                      tmp:final
                      "2009-11-11T17:00:00Z"^^xsd:dateTime ] .
```

Listing 5.3: A time-annotated triple using reification [4] representing the valid time in an interval-based representation.

```
:me foaf:workplaceHomepage <http://me.example.org/> _:c .
_:c tmp:interval [ tmp:initial
                    "2008-04-01T09:00:00Z"^^xsd:dateTime ;
                   tmp:final
                    "2009-11-11T17:00:00Z"^^xsd:dateTime ] .
```

Listing 5.4: A time-annotated triple using N-Quads [5].

relation to graphs. Alternatively, JSON Linked Data (JSON-LD) [50] can be used for formatting triples into named graphs.

5.4 Time Annotation

An example of equivalent time-annotated triples using the three annotation approaches (using the time vocabulary [31] introduced in previous work, see Subsection 2.2.5) can be found in respectively Listing 5.3, Listing 5.4 and Listing 5.5.

5.5 Conclusion

There are several reasons for not using reification as a method of annotating triples. The first reason is that the amount of triples in a dataset significantly increases when having a lot of reified triples. This is because each triple has to be split up into three separate triples, with an extra triple for each annotation. A second reason [51] is that these reifications lack decent semantics which also causes queries targeted to non-reified triples not being matched with reified triples (unless rule inferencing is in place).

```
foaf:workplaceHomepage#1 sp:singletonPropertyOf foaf:workplaceHomepage .
:me foaf:workplaceHomepage#1 <http://me.example.org/> .
foaf:workplaceHomepage#1 tmp:interval [
    tmp:initial "2008-04-01T09:00:00Z"^^xsd:dateTime ;
    tmp:final   "2009-11-11T17:00:00Z"^^xsd:dateTime ] .
```

Listing 5.5: A time-annotated triple using Singleton Properties [6] .

Singleton properties are an improvement of reification on the condition that these properties can be clustered, both on their object resources and metadata, otherwise we have the same problem as with reification.

Graphs are much more flexible than previous annotation types, the only disadvantage is that many existing RDF parsers are not forward compatible with this.

Chapter 6

Streaming RDF

This chapter is dedicated to the study of *continuous* queries over RDF data *streams*. Some applications require data to be continuously updated. This however, requires some changes or extensions to the existing SPARQL protocol.

The problem with the current (Semantic) Web, is that even though data is increasingly available, no system is sufficiently capable to answer questions that require systems to handle rapidly changing data [52]. Examples [52] of such questions are: Where are all my friends meeting? Which page on this website is the most popular? How many people are walking on this bridge that originate from York? Data Stream Management Systems (DSMSs) [53] can already analyze these types of questions, but they currently can not reason over this data and do not support the Linked Data protocols (RDF, SPARQL). Reasoners are much more processor heavy than DSMSs, because their reasoning is based on temporal logic and belief revision. This could be good for changes at a low frequency, but certainly not for high-speed streaming data. *Stream reasoning* is a research area [52, 8] that aims to integrate data streams and reasoners to solve these issues.

6.1 Stream Reasoning

This section will further explain the concept of Stream Reasoning based on previous work [52, 8].

Definition 4 (Stream Reasoning)**.** *Stream Reasoning is the logical reasoning in real time on gigantic and inevitably noisy data streams in order to support the decision process of extremely larger numbers of concurrent users.*

Stream processing makes use of the terms *window* [53] and *continuous processing* [54] which can be reused in the context of Stream Reasoning which will be explained hereafter. *Stepsize*, *time-annotation* and *query registration* are also frequently used concepts in the Stream Reasoning domain.

Window In Stream Reasoning, only a subset of facts which are most recently observed are used for reasoning instead of taking into account all the available information. The first reason for windowing this data is simply for saving computer resources to be able to respond to real time events. The second reason is that real-time applications mostly have the silent assumption that old information becomes irrelevant at a certain point, which will be a window edge.

Stepsize If there only was one defined window in a stream, then we would assume that the window edge always ends at the current moment. Meaning that if the window size would be 1 hour, that every possible time a query would be issued, all the data up until one hour before that would be taken into account. This requires the reasoner to constantly update the contents of this window. A stepsize prevents this continuous recalculation of the window contents by only moving the window at the frequency defined by that parameter. A stepsize of 15 minutes for example would mean that the window is advanced every 15 minutes.

Continuous Processing Most reasoning engines have processes which have a defined beginning and ending, initiating a request to the reasoner starts the process and it is ended when the result is returned. Stream Reasoning is not based on this processing life cycle, instead clients should register a query at the reasoner which then are continuously evaluated with constantly changing data.

Query Registration Continuous Processing is a requirement [52] for stream reasoners. All queries from a certain client must be registered to a streaming-SPARQL-endpoint once, and should continuously remain active. This way the server can immediately send new incoming query results to the client that issued the request. C-SPARQL [55] for example extends the SPARQL syntax to allow `REGISTER` clauses with some additional parameters about the desired update frequency. More details on this can be found in Subsection 6.2.1.

Time Annotation C-SPARQL [55], among others, extends the existing RDF triples to include timestamps. It does this by *annotating* triples by using a new tuple structure with a timestamp $\mathcal{T}$ on which the triple is valid. These stream triples are monotonically non-decreasing, but not strictly increasing, because there is no requirement for annotated triples to be unique in the stream. A formal representation of an RDF stream using these timestamps $\mathcal{T}$ can be found in Equation 6.1.

$$\begin{gathered}\dots \\ (\langle subj_i, pred_i, obj_i \rangle, \mathcal{T}_i) \\ (\langle subj_{i+1}, pred_{i+1}, obj_{i+1} \rangle, \mathcal{T}_{i+1}) \\ \dots\end{gathered} \tag{6.1}$$

There still exist a lot of issues and challenges [52] with Stream Reasoning. For example the problem of (partially) incomplete information, and how to handle this in real-time scenarios. Or the strict real-time constraints some systems require, which can currently not even always be guaranteed with the non-streaming engines. Finally, a complete formal theory of Stream Reasoning is still necessary with the requirements listed in related work [52].

6.2 SPARQL Streaming Extensions

6.2.1 C-SPARQL

Continuous SPARQL (C-SPARQL) [7] handles static and continuously changing data differently in the context of querying. Just like Data Stream Management Systems (DSMSs) [53], C-SPARQL requires registration of queries which will tell the server to return new results to the client as new data becomes available. The example "*How many people are walking on this bridge that originate from York?*" is a perfect example in the case of C-SPARQL. Here we would have an RDF stream "*Which people are walking on this bridge?*" and a static RDF part for "*Which people originate from York?*".

In brief, C-SPARQL will map its queries to an internal model that is transformed using specified methods to create queries that will intelligently distribute work between DSMSs and SPARQL engines. The following sections summarize previous work on C-SPARQL [7].

6.2.1.1 Additions to SPARQL

RDF Stream Data Type As explained in Section 6.1, an RDF stream is an ordered sequence of pairs that consist of triples and their corresponding timestamp $\mathcal{T}$.

Windows Data sources are identified using the `FROM STREAM` clause, instead of `FROM` that is used for static RDF stores. Section 6.1 already explained the concept of *windows*. These are selected from a stream as a `RANGE` parameter added to the new `FROM STREAM` parameter. This window can either be *physical* or *logical*, they respectively indicate a set number of triples and a time duration. Logical windows can be either *sliding* or *tumbling*, which respectively means that the window progresses in a given step size and the window progresses with the window size equal to the step size. An example of this new clause can be seen in Listing 6.2, and the exact syntax of this is defined in Listing 6.1.

Query Registration Because the output of the query needs to be continuously returned, a query registration must happen. The frequency at which the query is computed is determined by the server unless an optional `COMPUTED EVERY` clause is added

```
<from-stream-clause> ::= "FROM " <named> " STREAM" <IRI> "
                          [RANGE " <window> "]"
<named>              ::= "NAMED" | ""
<window>             ::= <logical-window> | <physical-window>
<logical-window>     ::= <number> <time-unit> <window-overlap>
<time-unit>          ::= "ms" | "s" | "m" | "h" | "d"
<window-overlap>     ::= "STEP " <number> <time-unit> | "TUMBLING"
<physical-window>    ::= "TRIPLES " number
```

Listing 6.1: The C-SPARQL `FROM STREAM` syntax.

```
PREFIX t: <http://linkedurbandata.org/traffic#>

SELECT DISTINCT ?tollgate ?passages
FROM STREAM <http://streams.org/citytollgates.trdf>
            [RANGE 10m STEP 1m]
WHERE {
    ?tollgate t:registers ?car .
}
```

Listing 6.2: Example [7] of the C-SPARQL `FROM STREAM` clause.

to `REGISTER QUERY`. The full syntax can be seen in Listing 6.3. An example of the query registration can be seen in Listing 6.4. The `REGISTER` clause on the first line is an indication to the engine that this is a continuous query. After that, `COMPUTE EVERY` tells with which frequency the query should be recomputed, this is in this case 15 minutes. Line 2 is a simple `SELECT` clause that is identical to regular SPARQL, as is the `FROM` on line 3. However, on line 4 we have a `FROM STREAM` clause. The previous `FROM` statement assumes that the URI refers to static data, but `FROM STREAM` handles the URI as a continuously updating stream. The `RANGE` clause on line 5 contains the window- and stepsize, which are in this case respectively 1 hour and 15 minutes.

Stream Registration Instead of registering queries, streams can also be registered. This can be used to create new streams from a certain query result. An important requirement for the registered query in this case, is that they only can be of the types `CONSTRUCT` or `DESCRIBE`. The reason for this is that the timestamp needs to encoded into RDF, which is not possible in raw triple output from a `SELECT` query. The syntax can be seen in Listing 6.5 and an example of this new clause is shown in Listing 6.6.

Multiple Streams SPARQL puts no restriction on the amount of streams that can be queried at once. This uses the same semantics as the normal SPARQL `GRAPH`s as explained in Subsection 2.3.1.

```
<registration>   ::= "REGISTER QUERY " <string> <computed-every> "
                      AS" <sparql-query>
<computed-every> ::= "COMPUTED EVERY " <number> <time-unit>
```

Listing 6.3: The C-SPARQL query registration syntax.

```
REGISTER QUERY NumberOfGiuliaFollowersWhoAreReadingBooks COMPUTE EVERY 15m
     AS
SELECT count(distinct ?user) as ?numberOfGiuliaFollowersReadingBooks
FROM <http://streamingsocialdata.org/followersNetwork>
FROM STREAM <http://streamingsocialdata.org/reading>
                [RANGE 1h STEP 15m]
WHERE {
    ?user :follows :Giulia .
    ?user :isReading ?x .
    ?x a :Book .
}
```

Listing 6.4: Register a C-SPARQL-query [8].

```
<registration>   ::= "REGISTER STREAM " <string> <computed-every>"
                      AS" <sparql-query>
<computed-every> ::= "COMPUTED EVERY " <number> <time-unit>
```

Listing 6.5: The C-SPARQL stream registration syntax.

```
REGISTER STREAM CarsEnteringCityCenterPerDistrict
         COMPUTED EVERY 5m AS
CONSTRUCT {
    ?district t:has-entering-cars ?passages .
}
FROM STREAM <http://streams.org/citytollgates.trdf>
            [RANGE 30m STEP 5m]
WHERE {
    ?tollgate t:registers ?car .
    ?district c:contains  ?street .
    ?tollgate c:placedIn ?street .
} AGGREGATE {(?passages,
              COUNT, {?district, ?tollgate, ?car})}
```

Listing 6.6: Register a C-SPARQL-stream-query [7].

Timestamp Function A new function is introduced that can return the timestamp of a given triple/resource. This new function takes one mandatory argument, the variable, and one optional argument, the stream URI, that is defined with the `GRAPH` clause. This can for example be used in the `FILTER` clause as follows: `FILTER(timestamp(?car1)> timestamp(?car2))`

6.2.1.2 Definitions

This part will formally define the semantics of C-SPARQL, which will be reused in other streaming engines as well. Refer to Subsections 2.2.1 and 2.3.4 for the basics of the notations that will be used here.

RDF Stream An *RDF Stream* is defined as $\mathcal{R} = \{(\langle subj, pred, obj\rangle, \mathcal{T}) \mid \langle subj, pred, obj\rangle \in ((\mathcal{U} \cup \mathcal{B}) \times \mathcal{U} \times (\mathcal{U} \cup \mathcal{B} \cup \mathcal{L})), \mathcal{T} \in \mathbb{T}\}$ with $\mathbb{T}$ the infinite set of timestamps.

Window A window ω can be *sliding* with *range* ρ and *step* σ.
A window is *tumbling* with a range ρ if it is sliding with range ρ and step $\sigma = \rho$.

Logical window $\omega_l(\mathcal{R}, t_i, t_f) = \{\langle s, p, o\rangle, \mathcal{T} \in \mathcal{R} \mid t_i < \mathcal{T} \leq t_f\}$

Physical window $\omega_p(\mathcal{R}, n) = \{\langle s, p, o\rangle, \mathcal{T} \in \omega_l(\mathcal{R}, t_i, t_f) \mid c(\mathcal{R}, t_i, t_f) = n\}$ with $c(\mathcal{R}, t_i, t_f)$ the amount of triples in $\mathcal{R}$ with a timestamp in range $(t_i, t_f]$.

Timestamp function $ts(v, \mathcal{P}) = max(\{x \mid t \in \mathcal{P} \wedge v \in dom(t) \wedge x \in TS_set(v, t)\})$, with $TS_set(v, t)$ the set of timestamps associated with a variable v and a triple pattern t.

6.2.1.3 Architecture

As mentioned before, the C-SPARQL architecture consists of a static knowledge engine for SPARQL combined with a DSMS. Previous work [7] mentions the use of *Sesame* as a SPARQL engine and *STREAM* [53] as a DSMS. The first module of this architecture is the *parser*, that parses the C-SPARQL query and passes the result on to the *orchestrator*. The orchestrator is the module that is responsible for delegation parts of the original query to the static knowledge engine and the DSMS. This is done by translating the query into a static and dynamic part. This delegation happens only at registration time of the query, after that, the static and dynamic engines can keep using the modified query input. Note that the delegation to the DSMS requires another translation step from RDF to relational info. This translation also exists the other way around, for transcoding the results of the stream manager back to RDF so that they

can be combined with the static RDF data. An overview of this architecture can be seen in Figure 6.1. A full example of this pipeline can be found in previous work [7].

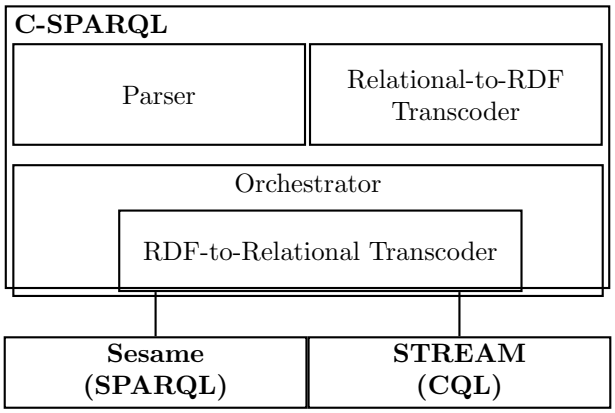


Figure 6.1: Overview of the C-SPARQL architecture.

This deep integration of a DSMS inside the SPARQL engine is an advantage of C-SPARQL compared to other approaches like Streaming SPARQL [15] or TA-SPARQL [11]. These other approaches store the stream before processing it, which introduces computational and storage overhead.

6.2.2 EP-SPARQL

Researchers [14] tried to close the gap between event processing systems and semantic reasoners. Event processing systems are capable of processing real-time event streams, but also only that. They are unable to integrate this real-time information with background knowledge. Semantic reasoning tools are only capable of processing static background knowledge, without those real-time events. Event Processing SPARQL (EP-SPARQL) [14] is introduced to combine the best of these two worlds. What makes EP-SPARQL different from other streaming SPARQL extensions, is that it is designed with the requirement of reasoning over *evolving knowledge*. This means that next to the traditional real-time queries like fetching the latest value of a given sensor, queries over the evolution of data are also possible. An example of such a query is getting a list of cities that are seeing a continuous drop in temperature during the last five days.

EP-SPARQL extends SPARQL by adding a set of binary operators and functions, which will be briefly explained in the following.

6.2.2.1 Functions

Three new functions are added that can be used inside filter expressions. The first function is `getDURATION()` which returns an `xsd:duration` literal that represents the length in time of the corresponding graph pattern. The functions `getSTARTTIME()` and

```
SELECT ?city
WHERE
    { ?city hasTemperature ?temp1 }
SEQ { ?city hasTemperature ?temp2 }
SEQ { ?city hasTemperature ?temp3
FILTER (
    ?temp3 < ?temp2 && ?price2 > ?price1
    && getDURATION() < "P3D"^^xsd:duration)
}
```

Listing 6.7: EP-SPARQL query asking for the cities with continuously dropping temperatures for the past three days using the `SEQ` operator.

`getENDTIME()` return an `xsd:dateTime` literal that respectively indicate the start and end date of the duration.

6.2.2.2 Operators

The binary operators `SEQ`, `EQUALS`, `OPTIONALSEQ` and `EQUALSOPTIONAL` add the ability to combine graph patterns with the focus on the *time* of the graphs.

`SEQ` allows you to join graph patterns in such a way that for example P_1 `SEQ` P_2 indicates that pattern P_1 must happen strictly before P_2, and these graphs will be joined if this condition is met. This behavior can be illustrated with the previously mentioned example of continuously dropping temperatures of cities, the associated query can be found in Listing 6.7.

The `EQUALS` operator can be used to indicate that graph patterns must happen at exactly the same time, and joins the graphs in that case.

The operators `OPTIONALSEQ` and `EQUALSOPTIONAL` work in a similar way, each combined with the `OPTIONAL` behavior of SPARQL.

6.2.2.3 Architecture

The EP-SPARQL architecture is based on event-driven backward chaining (EDBC) rules [56]. All queries are fully, both the static and dynamic parts, converted into EDBC rules in ETALIS. ETALIS [1] is a Prolog-based event processing framework which allows for event-driven reasoning on these events. This unified execution of both the static and dynamic parts of the query is a big advantage of EP-SPARQL when compared to related approaches, this removes the overhead of the communication between background knowledge systems and streaming systems as is the case with C-SPARQL.

[1] `https://code.google.com/p/etalis/`

6.2.3 SPARQLStream

An ontology-based streaming data access service [9] is explained that can be queried using the SPARQL$_{Stream}$ SPARQL extension. This approach is based on the mapping definition language Relational-to-Ontology (R_2O) [57] which allows conversion of data sources. Similarly to R_2O, a Stream-to-Ontology (S_2O) mapper is introduced. Just like in R_2O, S_2O allows you to define a set of mappings in terms of selections and transformations of the data source, which is in this case a stream. Each ontology requires it's own S_2O rules, which can become quite difficult when querying many different data stores.

A distinction of two types of streams is made [9]:

- **Event streams**: Streams in which tuples are generated at a variable rate.
- **Acquisitional streams**: Streams in which tuples are generated at a predefined regular interval.

This approach focuses on the applications where older tuples become irrelevant once newer tuples are generated, in contrast with the evolving knowledge requirement of EP-SPARQL. The main difference between this approach and C-SPARQL, is that the results of SPARQL$_{Stream}$ queries are windows of triples which are defined in time, instead of continuous streams.

6.2.3.1 Architecture

The system can accept queries using the SPARQL$_{Stream}$ language which extends SPARQL by adding operators over streams. The query is transformed to the query language for streams `SNEE`ql [58], using predefined S_2O mappings. The reason for using `SNEE`ql, is that it is able to query over both static and dynamic data. After the query processing in `SNEE`ql, the resulting tuples are transformed to RDF triples, which is what the client expects.

An overview of this architecture can be seen in Figure 6.2.

The two-way conversion between streams (in `SNEE`ql) and windows (in RDF) is an essential part of this architecture. The windowing of streams is done in a similar fashion as C-SPARQL, but with a slightly different syntax. This syntax being: `FROM start TO end [SLIDE int unit]`. The stream to window conversion can occur in three different modes that can be specified in the query definition: *a*) `RSTREAM` adds all tuples to the window that appear in the stream, *b*) `ISTREAM` only adds those tuples to the window that are new since the previous evaluation of the window and *c*) `DSTREAM` only adds the tuples that have been deleted stream to the window.

An example of a SPARQL$_{Stream}$ query can be found in Listing 6.8.

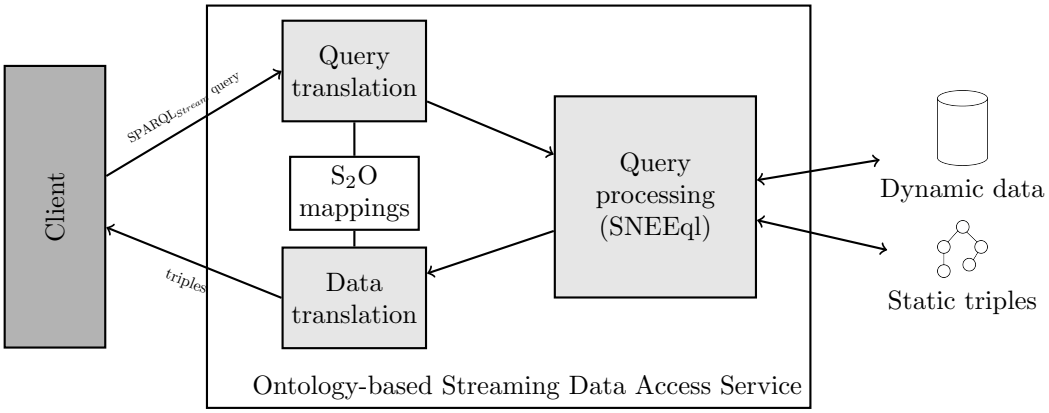


Figure 6.2: Overview of the SPARQL$_{Stream}$ architecture.

6.2.4 Streaming SPARQL

The Streaming SPARQL [15] approach goes to the foundations of SPARQL to add stream processing capabilities by extending the logical SPARQL algebra. An algorithm is also provided which can transform Streaming SPARQL queries to that extended algebra.

Streaming SPARQL is very similar to that of C-SPARQL, with those differences that *a*) streaming queries do not need to be registered explicitly, so no `REGISTER` operator exists, and *b*) the appendix of `FROM STREAM` is different, in this case it takes a `WINDOW` parameter for which the syntax can be found in Listing 6.9. The physical window and logical window are equivalent to the ones from C-SPARQL, respectively meaning that each window contains a given number of triples and each window is defined by a given range and step (slide) size. In fact, the fixed window from Streaming SPARQL is just a logical window with a step size equal to the range, this is equivalent to the tumbling window from C-SPARQL. This means that C-SPARQL and Streaming SPARQL are semantically equivalent, except for the fact that C-SPARQL queries also take a parameter in the `REGISTER` clause that defines at which interval the query needs to be executed, while in this case that parameter is automatically derived from the step size.

An example equivalent to the C-SPARQL example from Listing 6.4 can be found in Listing 6.10.

6.2.5 CQELS

Continuous Query Evaluation over Linked Stream (CQELS) [10] distances itself from the “black box” approaches like C-SPARQL and EP-SPARQL by using a “white box” approach. This means that CQELS implements all the query operators natively (white box) without transforming it first to another system (black box) which results in a lot less overhead. This approach also makes use of dynamic query rewriting to further

```
PREFIX fire: <http://www.semsorgrid4env.eu#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT RSTREAM ?WindSpeed
FROM STREAM <www.semsorgrid4env.eu/SensorReadings.srdf>
    [FROM NOW - 10 MINUTES TO NOW STEP 1 MINUTE]
FROM STREAM <www.semsorgrid4env.eu/SensorArchiveReadings.srdf>
    [FROM NOW - 3 HOURS TO NOW STEP 1 MINUTE]
WHERE {
    GRAPH <www.semsorgrid4env.eu/SensorReadings.srdf> {
        ?WindSpeed a fire:WindSpeedMeasurement ;
                   fire:hasSpeed ?speed .
    }
    SELECT AVG(?archivedSpeed) AS ?WindSpeedHistoryAvg
    WHERE {
        GRAPH <www.semsorgrid4env.eu/SensorArchiveReadings.srdf> {
            ?ArchWindSpeed a fire:WindSpeedMeasurement ;
                           fire:hasSpeed ?archivedSpeed .
        }
    } GROUP BY ?ArchWindSpeed
    FILTER (?WindSpeedAvg > ?ArchWindSpeed)
}
```

Listing 6.8: A SPARQL$_{Stream}$ query which checks the speeds of each sensor of the past ten minutes that exceed all the speeds of the past three archived hours [9].

```
<window>           ::= <logical-window> | <fixed-window>
                       | <physical-window>
<logical-window>   ::= "WINDOW" "RANGE" <valspec> "SLIDE"
                       <optional-valspec>
<fixed-window>     ::= "WINDOW" "RANGE" <valspec> "FIXED"
<physical-window>  ::= "WINDOW" "ELEMS" <number>
<valspec>          ::= <number> <time-unit>
<time-unit>        ::= "MS" | "S" | "MINUTE" | "HOUR" | "DAY" | "WEEK" |
    ""
<optional-valspec> ::= <valspec> | ""
```

Listing 6.9: The Streaming SPARQL `WINDOW` syntax.

```
SELECT count(distinct ?user) as ?numberOfGiuliaFollowersReadingBooks
FROM <http://streamingsocialdata.org/followersNetwork>
FROM STREAM <http://streamingsocialdata.org/reading>
                WINDOW 1 HOUR SLIDE 15 MINUTE
WHERE {
    ?user :follows :Giulia .
    ?user :isReading ?x .
    ?x a :Book .
}
```

Listing 6.10: A Streaming SPARQL-query equivalent to the C-SPARQL one in Listing 6.4.

```
<stream-graph-pattern> ::= "STREAM" "[" <window> "]" <var-or-iri>
                           "{" <triples-template> "}"
<window>               ::= <range> | <triple> | "NOW" | "ALL"
<range>                ::= "RANGE" <valspec> <optional-slide>
<triple>               ::= "TRIPLES" <number>
<valspec>              ::= <number> <time-unit>
                           | <valspec> <valspec>
<time-unit>            ::= "d" | "h" | "m" | "s" | "ms"
                           | "ns" | ""
<optional-slide>       ::= "SLIDE" <valspec> | ""
```

Listing 6.11: The CQELS stream graph pattern syntax.

```
SELECT ?locName ?locDesc
FROM NAMED <http://deri.org/floorplan/>
WHERE {
    STREAM <http://deri.org/streams/rfid> [NOW] {
        ?person lv:detectedAt ?loc .
    }
    GRAPH <http://deri.org/floorplan/> {
        ?loc lv:name ?locName. ?loc lv:desc ?locDesc .
    }
    ?person foaf:name "Bob Peters" .
}
```

Listing 6.12: CQELS query to get the name and description of the current location of Bob [10].

increase the performance of the engine. By using *dictionary encoding* [59], the engine can efficiently store much more data in the machine's memory for faster processing. This combined with a caching and indexing system allows for very fast data look-ups.

The syntax of CQELS is very similar to that of C-SPARQL and Streaming SPARQL, with that difference that the streams must be declared as graph patterns with an IRI as identification, the full syntax can be found in Listing 6.11. An example of a simple query can be found in Listing 6.12.

Results [10] show that CQELS performs a lot better than C-SPARQL and EP-SPARQL for large datasets, in some cases with large datasets this engine is 700 times faster than the two others. The white box approach is the main contributor for these results. Only for simple queries and small datasets C-SPARQL and EP-SPARQL could sometimes outperform CQELS.

6.2.6 TA-SPARQL

A paper [11] introduced Time-Annotated RDF (TA-RDF) together with the corresponding SPARQL extension Time-Annotated SPARQL (TA-SPARQL).

TA-RDF is based on the idea of time-annotating resources instead of the full triple. This allows for a representation inside native RDF. These resources are represented in

```
SELECT sum(?rain)
WHERE {
    <urn:OHARE> <urn:hasRainSensor>
        ?x["2009-01-01Z-06:00"^^xsd:date ..
           "2009-01-31Z-06:00"^^xsd:date] .
    ?x <urn:hasReading> ?rain .
}
```

Listing 6.13: TA-SPARQL query to find the total amount of rain in Chicago in January 2009 [11].

regular triples with the semantics that they are all part of a stream, so the resources are *frames* of the stream. The new property `dsv:belongsTo` is added to these resources to indicate to which stream this resource belongs to. The property `dsv:hasTimestamp` indicates the exact timestamp of that resource inside the stream, this can be a literal or the `dsv:Nil` resource for when the timestamp is unavailable. A frame having the `dsv:belongsTo` property, implies that this subject inherits all the properties of the target stream. A frame is also restricted to be part of only one stream with only one timestamp.

The TA-SPARQL extension adds a set of new constructs that can be used to very easily query over a range of time or at specific moments in time. All of the additions can be translated to an equivalent SPARQL 1.1 representation. The new constructs are briefly explained here, the details of the transformation to native SPARQL can be found in related work [11]. Subjects in TA-SPARQL can be appended with a date or time range between square brackets to look for certain triple patterns. An example of such a query can be found in Listing 6.13.

The implementation of the TA-SPARQL query engine was done in a black box way, by delegating the static part of the query to a simple RDF store and the dynamic part to a usecase-specific stream indexer.

6.2.7 Conclusion

Based on previous work [60, 61, 10] and the SPARQL extension summaries in the previous subsections, Table 6.1 shows a brief overview of the most important characteristics of these extensions.

Approach	**Characteristics**
C-SPARQL	• Query registration • Time windows • **Black box** query processing with STREAM and Sesame • Reasoning over **temporal** knowledge
EP-SPARQL	• Event processing • No time windows • Immediate event processing at occurrence time • **Black box** query processing with ETALIS • Reasoning over **temporal** and **evolving** knowledge
SPARQL$_{Stream}$	• Stream-to-Ontology (S_2O) mapping rules • Time windows • **Black box** query processing with `SNEE`ql • Reasoning over **temporal** knowledge
Streaming SPARQL	• Query registration • Time windows • **White box** query processing by extending logical SPARQL algebra • Reasoning over **temporal** knowledge
CQELS	• Time windows • **White box** query processing by native implementation • Significant performance advantage
TA-SPARQL	• Direct access to time annotations • Implicit time windows • **Black box** query processing with RDF store and custom stream indexer • Reasoning over **temporal** and **evolving** knowledge

Table 6.1: Overview of the most important characteristics for some streaming SPARQL extensions.

Chapter 7

Dynamic Data Representation

7.1 Temporal Domains for Timeliness

This section will discuss different ways on how *temporal domains*[1] can be used to reach a sufficient data timeliness for representing the volatility of data. This section will focus on the semantics of representing this frequency, while the next section will go deeper into the syntactical details. Several types of metadata for representing data volatility will be compared. First, a very generic but complex approach will be explained. After that, two more simple approaches of temporal domains [31, 4] will be explained in Subsection 2.2.5. At the end of this chapter, a conclusion will be made about which approach is the best for this research.

As was already explained in Subsection 2.2.5, *versioning* of the complete graph might be not the best choice for annotating data in this context, so the idea of *time labeling* for certain triples will be used.

7.1.1 Volatility Pattern Formula

A very generic approach for representing the volatility of certain data is by using a mathematical formula which can be evaluated to find the next the data update time. The data provider would have to annotate dynamic data with such a formula. The client would then evaluate that function and find out when the data will be updated. Examples of function arguments could be the current client time, amount of requested updates on that data element and elapsed time since the first data retrieval. These types of function arguments could differ in other use cases and might even be declared inside that same annotation.

This approach is very generic and could be used in many different use cases, it is however

[1] Temporal domains in this work refer to methods for representing the dynamism of data.

too complex for our solution. There are several reasons for not using this approach for our use case.

a) Finding a volatility pattern formula for a data element can be very complex for the server. Advanced pattern matching algorithms might be required in some cases, which are not within the scope of this work. *b*) Evaluation at client side can also become very complex. Depending on the desired form of the function, evaluation algorithms such as root-finding might be required. This evaluation can become too expensive for the client when they have limited computational power, or when the evaluation must be done at a high frequency. *c*) Declaring and interpreting the functions as part of an annotation can also be non-trivial. Since we are working in RDF datasets, ontologies based on MathML [62] or OpenMath [63] could be a solution to this.

While this approach is very generic and might be an interesting future research topic, it is currently too complex for our desired use case. So more simple, but still sufficiently expressive approaches will be discussed in following sections.

7.1.2 Time Interval and Expiration Time

Subsection 2.2.5 explained the difference between *valid* and *transaction* times. We can use the *valid* times to indicate in which time interval the facts are valid in the world. We refer to these *valid* times using *interval-based* labeling, where facts are annotated with an *initial* and *final* time. *Interval-based* labeling can be desired when we need multiple instances of the same fact, possibly with different values, that are valid in their own time range. This type is required for scenarios where historical data is important, for example when the complete history of delays in a certain train station is required for some kind of reasoning.

Note that the term *fact* is here always used to refer to all versions of a certain knowledge element. For example, the dynamic fact "delay of departure with id `train1234`" contains all the versions of that fact each having different time intervals. This is slightly different than the RDF definition of *fact*, which just refers to a triple. In the remainder of this document, the former definition will be used.

Instead of using *interval-based* labeling, we can also use *point-based* labeling. *Point-based* labeling is based on the *transaction* times explained in Subsection 2.2.5 where each fact version is annotated with just one timestamp, in this case the moment in time this version expires. This type of annotation is advantageous in cases where only one instance of a fact is available in a dataset at a given time. This would require the data provider to constantly overwrite that fact with its new value and expiration time.

Annotation with the interval approach requires two elements for each dynamic resource, the initial and final time of the interval. Annotation with the expiration approach only needs the expiration time to be appended to the fact. When using time intervals, the data provider is responsible for making sure that when each interval expires, a new consecutive interval is created. Dynamic facts that have no valid interval at a given time are undefined for that time. In case of the expiration times, the data provider is

again responsible for the addition of new expiration times to avoid undefined facts at a given time. The expiration times can be consecutive if the semantics of the fact indicate that the previous expiration time should be interpreted as the beginning of valid time interval, and the current expiration time as the end of this interval.

7.1.3 Influence of Data Volatility

The sole existence of the last known fact value when using expiration times can be seen as a disadvantage because of the loss of history. But this is at the same time a big advantage when this dynamic data is very volatile. When using time intervals for many volatile facts, a lot of consecutive interval-based facts would quickly accumulate. Without a decent way of removing or aggregating old data, this will eventually result in an unmanageably large dataset, causing slow query executions. When using expiration times, the data volatility will have no influence on the size of the dataset, since the facts will always be overwritten when the old versions expire.

7.1.4 Conclusion

As was already mentioned, the first approach that is based on a mathematical formula is considered too complex for our research. Both interval-based and pointed-based time labeling are fitting solutions for our problem, we will refer to these two as types of temporal domain.

7.2 Methods for Time Annotation

While the previous section focussed on the semantics of time annotation, this section will explain the syntactical details. Several types of time annotation will be discussed here. This chapter will build further upon the annotation concepts that were explained in Chapter 5.

All of the following types will be illustrated by declaring dynamic information about certain train departures, more specifically the platform each train departs from. This will each time be done for both the time interval and expiration time approach. The timeliness of the fact is assumed to be one minute in these cases, but this could be any frequency and must not be uniform across time and triples. After this minute, the data provider has to make sure that a new time-annotated fact is available with a time interval or expiration time of a consecutive minute containing the platform at that time, unless of course the train has already departed. This mechanism doesn't actually change after the train departure, but the expiration time becomes infinity.

The train platforms will be declared in the form of the triple `?id t:platform ?platform` with the variable `?id` containing a unique URI for a train departure and `?platform` containing a platform name encoded as a string. Listing 7.1 shows the

```
@prefix t: <http://example.org/train> .
@prefix departure: <http://example.org/traindata/departures> .
@prefix platform: <http://example.org/traindata/platform> .
@prefix tmp: <http://example.org/temporal> .

departure:4815  t:platform platform:1a .
departure:1623  t:platform platform:8 .
departure:42108 t:platform platform:12 .
```

Listing 7.1: Original train platform triples before time annotation is added to them.

triples which will be used as a source of train platform information. These triples are a snapshot of a certain unknown moment in time. Time information will be added using various ways of annotation, some of which will be discussed in the following subsections.

Each of these subsections will contain a triple-count function $f(t)$ showing the amount of triples required to represent this data with its annotation in function of the original amount of triples all facts, t. For the example the Listing 7.1 contains three triples, so this original value t would be three in this case. Note that this function is limited to one fact, if we would annotate two different facts with time, two separate functions would be needed to calculate the amount of triples required.

7.2.1 Reification

This first type of annotation is considered to be the least efficient way of adding information to facts in terms of triple count. The original triples are reified using a unique subject for each triple, possibly by using a blank node. This subject is then used to add additional information to, this would be either a time interval or an expiration time.

Listing 7.2 and Listing 7.3 show the reified form of the triples from Listing 7.1 using respectively time interval annotation and expiration times.

The major disadvantage of reification is the large number of of triples that is required to represent a reified fact. This reification must be done for each fact, resulting in a large overhead of shared information that could be avoided when using a different type of annotation. In this case we need five triples per fact when using interval-based annotation and four triples when using expiration times, which is in either case a huge increase of triples. When using this type of annotation for all departures in every possible train station, this can quickly become very large.

Listing 7.4 shows a better way for the time interval representation. In this case, the intervals of the different reified triples can be grouped together so that the interval only has to be declared once. This results in the same amount of triples as with the expiration times plus two triples. This doesn't take away the fact that four triples per fact is still a lot.

The issue with volatile data as was explained in Section 7.1.3 is severly worse when

```
_:stmt1 rdf:subject departure:4815 .
_:stmt1 rdf:predicate t:platform .
_:stmt1 rdf:object platform:1a .
_:stmt1 tmp:intervalInitial "2014-10-02T12:00:00Z"^^xsd:dateTime .
_:stmt1 tmp:intervalFinal "2004-04-12T12:01:00Z"^^xsd:dateTime .

_:stmt2 rdf:subject departure:1623 .
_:stmt2 rdf:predicate t:platform .
_:stmt2 rdf:object platform:8 .
_:stmt2 tmp:intervalInitial "2014-10-02T12:00:00Z"^^xsd:dateTime .
_:stmt2 tmp:intervalFinal "2004-04-12T12:01:00Z"^^xsd:dateTime .

_:stmt3 rdf:subject departure:42108 .
_:stmt3 rdf:predicate t:platform .
_:stmt3 rdf:object platform:12 .
_:stmt3 tmp:intervalInitial "2014-10-02T12:00:00Z"^^xsd:dateTime .
_:stmt3 tmp:intervalFinal "2004-04-12T12:01:00Z"^^xsd:dateTime .
```

Listing 7.2: Dynamic train platform triples using reification with time intervals.

```
_:stmt1 rdf:subject departure:4815 .
_:stmt1 rdf:predicate t:platform .
_:stmt1 rdf:object platform:1a .
_:stmt1 tmp:expiration "2014-10-02T12:01:00Z"^^xsd:dateTime .

_:stmt2 rdf:subject departure:1623 .
_:stmt2 rdf:predicate t:platform .
_:stmt2 rdf:object platform:8 .
_:stmt2 tmp:expiration "2014-10-02T12:01:00Z"^^xsd:dateTime .

_:stmt3 rdf:subject departure:42108 .
_:stmt3 rdf:predicate t:platform .
_:stmt3 rdf:object platform:12 .
_:stmt3 tmp:expiration "2014-10-02T12:01:00Z"^^xsd:dateTime .
```

Listing 7.3: Dynamic train platform triples using reification with a expiration times.

```
_:stmt1 rdf:subject departure:4815 .
_:stmt1 rdf:predicate t:platform .
_:stmt1 rdf:object platform:1a .
_:stmt1 tmp:interval _:interval1

_:stmt2 rdf:subject departure:1623 .
_:stmt2 rdf:predicate t:platform .
_:stmt2 rdf:object platform:8 .
_:stmt1 tmp:interval _:interval1

_:stmt3 rdf:subject departure:42108 .
_:stmt3 rdf:predicate t:platform .
_:stmt3 rdf:object platform:12 .
_:stmt1 tmp:interval _:interval1

_:interval1 tmp:intervalInitial "2014-10-02T12:00:00Z"^^xsd:dateTime .
_:interval1 tmp:intervalFinal "2004-04-12T12:01:00Z"^^xsd:dateTime .
```

Listing 7.4: Alternative representation of the dynamic train platform triples using reification with time intervals.

```
<http://example.org/sp1> sp:singletonPropertyOf t:platform .
<http://example.org/sp1> tmp:intervalInitial "2014-10-02T12:00:00Z"^^xsd:
    dateTime .
<http://example.org/sp1> tmp:intervalFinal "2004-04-12T12:01:00Z"^^xsd:
    dateTime .

departure:4815  <http://example.org/sp1> platform:1a .
departure:1623  <http://example.org/sp1> platform:8 .
departure:42108 <http://example.org/sp1> platform:12 .
```

Listing 7.5: Dynamic train platform triples using singleton properties with time intervals.

```
<http://example.org/sp1> sp:singletonPropertyOf t:platform .
<http://example.org/sp1> tmp:expiration "2014-10-02T12:01:00Z"^^xsd:
    dateTime .

departure:4815  <http://example.org/sp1> platform:1a .
departure:1623  <http://example.org/sp1> platform:8 .
departure:42108 <http://example.org/sp1> platform:12 .
```

Listing 7.6: Dynamic train platform triples using singleton properties with a expiration times.

using reification. Because with interval-based annotation not only one triple would be created when an interval expires, but five new triples would be added for each dynamic fact.

The triple-count functions for reification annotation can be found in 7.1.

$$f_{R\_interval}(t) = 5 * t \tag{7.1a}$$
$$f_{R\_expiration}(t) = 4 * t \tag{7.1b}$$
$$f_{R\_interval\_better}(t) = 4 * t + 2 \tag{7.1c}$$

7.2.2 Singleton Properties

Singleton properties were introduced as an improvement to the reification approach. In this case, singleton properties are created for each of the dynamic triple predicates. As was noted in Chapter 5, creating singleton properties for each relation introduces overhead and should be avoided. This is we created one singleton property which is reused for each annotation of that type.

Listing 7.5 and Listing 7.6 respectively show the singleton properties transformation of the triples from Listing 7.1 using time interval annotation and expiration times.

As can be seen in these examples, fewer triples are required to declare the same information as with reification. Only one extra triple is required for the entire block when

```
<http://example.org/graph1>
{
    departure:4815  t:platform platform:1a .
    departure:1623  t:platform platform:8 .
    departure:42108 t:platform platform:12 .
}

<http://example.org/graph1> tmp:intervalInitial "2014-10-02T12:00:00Z"^^
    xsd:dateTime .
<http://example.org/graph1> tmp:intervalFinal "2004-04-12T12:01:00Z"^^xsd:
    dateTime .
```

Listing 7.7: Dynamic train platform triples using explicit graphs with time intervals.

compared to the original static triples, excluding the actual temporal domain triples. But it should be noted that this form of "compression" can only be achieved within clusters of dynamic triple that have the same predicates. If we would for example in this case also wish to add the dynamic facts of train delays, another separate singleton property would be required.

There is however another improvement possible. Even though only two or three triples are required to add the actual annotation information, the singleton property itself still has to be declared as well. So this singleton property declaration is purely overhead when compared to the solutions discussed in next sections.

The triple-count functions for singleton property annotation can be found in 7.2.

$$f_{SP\_interval}(t) = t + 3 \tag{7.2a}$$
$$f_{SP\_expiration}(t) = t + 2 \tag{7.2b}$$

7.2.3 Explicit Graphs

This and next section is based on the idea of graphs. This section uses regular graphs as they were introduced in RDF 1.1.

Listing 7.7 and Listing 7.8 respectively show the explicit graph approach using time interval annotation and expiration times in the *TriG* [26] syntax.

This approach is very similar to the singleton properties annotation type. Instead of annotating the common predicate of the triples, the context of the triples is annotated. This solves the overhead problem of adding a separate triple to declare the singleton property.

Even though this is a very concise and easy method of annotating data, there is still the fact that dynamic triples have to be declared within a certain context, which requires prior knowledge of this context. That is what the approach explained in next section will try to circumvent.

```
<http://example.org/graph1>
{
    departure:4815  t:platform platform:1a .
    departure:1623  t:platform platform:8 .
    departure:42108 t:platform platform:12 .
}

<http://example.org/graph1> tmp:expiration "2014-10-02T12:01:00Z"^^xsd:
    dateTime .
```

Listing 7.8: Dynamic train platform triples using explicit graphs with a expiration times.

The triple-count functions for explicit graph annotation can be found in 7.3.

$$f_{EG\_interval}(t) = t + 2 \tag{7.3a}$$
$$f_{EG\_expiration}(t) = t + 1 \tag{7.3b}$$

7.2.4 Implicit Graphs

Up until now, all types of annotation discussed in this chapter are supported at SPARQL endpoints adhering to RDF 1.1. If it is taken into account that a TPF endpoint is used, another interesting way of representing graphs becomes possible.

Triple Pattern Fragments can be selected by subject, predicate and object. This means that each triple can be found by using the elements of itself for finding the Triple Pattern Fragment of itself. This implies that each triple has in fact a unique URI when using TPF. And thereby each triple has a unique implicit graph attributed to it, which is identified by a URI that can be used in other triples.

Asuming there is a TPF interface at `http://example.org/endpoint/train`, Listing 7.9 and Listing 7.10 respectively shows the implicit graph approach using time interval annotation and expiration times.

Graphs are now implicitly defined, but now there is the problem that each of these implicit graphs must have their own separate time annotation, since these implicit graphs cannot as easily be grouped anymore. This is however something that might be interesting to research further in the future, a possible solution could intelligently make use of HTTP redirects for allowing implicit graph clustering.

A major advantage of this approach when compared to all previous annotation types is the fact that this is fully compatible with clients who are not aware of the time annotations of these dynamic triples. This is because the triples of Listing 7.1 are exactly available in the Listings 7.9 and 7.10. This means that regular clients can query this data and will assume that this is static data, which might be focused over the alternative where this data is encapsulated in some other triple structure as is the case

```
departure:4815  t:platform platform:1a ?graph1 .
departure:1623  t:platform platform:8 ?graph1 .
departure:42108 t:platform platform:12 ?graph1 .

<http://example.org/endpoint/train?subject=http://example.org/traindata/
    departures/4815&predicate=http://example.org/train/platform&object=
    http://example.org/traindata/platform/1a> tmp:intervalInitial
    "2014-10-02T12:00:00Z"^^xsd:dateTime .
<http://example.org/endpoint/train?subject=http://example.org/traindata/
    departures/4815&predicate=http://example.org/train/platform&object=
    http://example.org/traindata/platform/1a> tmp:intervalFinal
    "2004-04-12T12:01:00Z"^^xsd:dateTime .

<http://example.org/endpoint/train?subject=http://example.org/traindata/
    departures/1623&predicate=http://example.org/train/platform&object=
    http://example.org/traindata/platform/8> tmp:intervalInitial
    "2014-10-02T12:00:00Z"^^xsd:dateTime .
<http://example.org/endpoint/train?subject=http://example.org/traindata/
    departures/1623&predicate=http://example.org/train/platform&object=
    http://example.org/traindata/platform/8> tmp:intervalFinal "2004-04-12
    T12:01:00Z"^^xsd:dateTime .

<http://example.org/endpoint/train?subject=http://example.org/traindata/
    departures/42108&predicate=http://example.org/train/platform&object=
    http://example.org/traindata/platform/12> tmp:intervalInitial
    "2014-10-02T12:00:00Z"^^xsd:dateTime .
<http://example.org/endpoint/train?subject=http://example.org/traindata/
    departures/42108&predicate=http://example.org/train/platform&object=
    http://example.org/traindata/platform/12> tmp:intervalFinal
    "2004-04-12T12:01:00Z"^^xsd:dateTime .
```

Listing 7.9: Dynamic train platform triples using implicit graphs with time intervals.

```
departure:4815  t:platform platform:1a ?graph1 .
departure:1623  t:platform platform:8 ?graph1 .
departure:42108 t:platform platform:12 ?graph1 .

<http://example.org/endpoint/train?subject=http://example.org/traindata/
    departures/4815&predicate=http://example.org/train/platform&object=
    http://example.org/traindata/platform/1a> tmp:expiration "2014-10-02
    T12:01:00Z"^^xsd:dateTime .

<http://example.org/endpoint/train?subject=http://example.org/traindata/
    departures/1623&predicate=http://example.org/train/platform&object=
    http://example.org/traindata/platform/8> tmp:expiration "2014-10-02T12
    :01:00Z"^^xsd:dateTime .

<http://example.org/endpoint/train?subject=http://example.org/traindata/
    departures/42108&predicate=http://example.org/train/platform&object=
    http://example.org/traindata/platform/12> tmp:expiration "2014-10-02
    T12:01:00Z"^^xsd:dateTime .
```

Listing 7.10: Dynamic train platform triples using implicit graphs with a expiration times.

	Triple-count	Quads	TPF	Backwards compatible
Reification	$f_{R\_interval}(t) = 5 * t$ $f_{R\_expiration}(t) = 4 * t$ $f_{R\_interval\_better}(t) = 4 * t + 2$	no	no	no
Singleton Properties	$f_{SP\_interval}(t) = t + 3$ $f_{SP\_expiration}(t) = t + 2$	no	no	no
Explicit Graphs	$f_{EG\_interval}(t) = t + 2$ $f_{EG\_expiration}(t) = t + 1$	required	no	no
Implicit Graphs	$f_{IG\_interval}(t) = t + 2$ $f_{IG\_expiration}(t) = t + 1$	no	yes	yes

Table 7.1: Overview of the most important characteristics of the different annotation types. The column *triple-count* contains the triple-count functions in terms of the original required amount of triples t. *Quads* indicates whether the annotation type requires the concept of quads. *TPF* indicates if the annotation type requires a Triple Pattern Fragments interface. The last column indicates whether or not the annotation type allows regular clients to retrieve the dynamic facts as static data.

with the previous annotation types. Clients who are able to detect them as dynamic triples will be able to retrieve their time annotation.

The triple-count functions for implicit graph annotation can be found in 7.4.

$$f_{IG\_interval}(t) = t + 2 \tag{7.4a}$$
$$f_{IG\_expiration}(t) = t + 1 \tag{7.4b}$$

7.2.5 Conclusion

From Table 7.1 it is clear that reification should be avoided in any possible scenario because of the large triple count. If the need exists to support RDF 1.0, singleton properties will be the best solution as they can be encoded using only triples. If compatibility is required for clients who are not aware of these time annotations, implicit graphs should be used, assuming a TPF endpoint is used. If neither of these requirements are applicable, graphs might be used.

Chapter 8

Solution

This chapter will explain the details of the proposed solution to formulate an answer to the research questions. First, the required changes to the TPF server will be explained, more specifically the changes required to the data model. After that, the architecture of the query streamer will be explained in a generic way. Meaning that this architecture can be used for any type of temporal domain and time annotation as presented in Chapter 7. Finally, some important remarks on this solution requiring some additional attention are discussed.

Our solution requires an extra software layer on top of the Triple Pattern Fragment client to be able to handle dynamic triples. The Triple Pattern Fragment server only requires a very small addition in case a graph-based annotation approach is used. This is because at the time of writing, the TPF server and client implementation do not support graphs.

8.1 Server Time Annotation

Data providers wishing to offer dynamic, i.e. time-sensitive triples, are required to first choose the temporal domain for these triples. If multiple temporarily ordered versions of the same fact are of interest, the interval-based approach should be used. When only the last version of a certain fact is required, time annotation by expiration times should be used. This is because time interval annotation allows you to declare multiple versions of the same fact with a different time range, while expiration times contain less information about this time range which makes them unsuitable for declaring multiple versions. The latter will have a constant amount of fact versions as long as the amount of different facts stays constant, while the former will result in a continuously growing dataset, possibly requiring some way of mitigating this continuous growth.

Depending on the data volatility, an update frequency has to be determined to update the dynamic facts to achieve a desired level of data currency. One could synchronize

the timeliness with its volatility if this frequency is somehow predictable. For example, temperature sensors performing measurements each minute can easily be synchronized with the update frequency in the dataset that will store this data. If the data volatility can not be predicted, or is too complex, an update frequency could be determined by using statistical models. In some cases, the data volatility is unbounded, meaning that the data is different every time a measurement is done. In which case the update frequency can be altered depending on the temporal importance. For example in case of the train delay updates, an exponential increase of updates could be done when the train departure is nearing, with a complete stop of updates after that train has finally departed.

The next choice the data provider has to make is which implementation of time annotation they will use. This choice will have an influence on both the triple storage efficiency and the technology required to query this data.

8.2 Query Streamer

The query streamer is an extra layer on top of the Triple Pattern Fragments client. No changes are required to the basic TPF client, except for the addition `GRAPH` support when the graph annotation approach is in use.

This extra layer acts as a proxy for executing queries. The client can execute regular SPARQL queries against this layer. In case the query streamer detects that the given query requires result streaming, the results will be streamed to the client, otherwise the single static query results will be given to the client only once.

When the user sends a query to this client, the **rewriter** module will split up this query into a static and dynamic part using metadata from the triple patterns. The dynamic part is then forwarded to the **streamer** module, the static part is stored for later usage.
The **streamer** module is able to execute a dynamic query, after which the results are sent to the **time filter**.
The **time filter** analyses all dynamic query results, and selects only those that are *currently active*.[1] Using these filtered results, a new execution time for the dynamic query is calculated and a new callback to the **streamer** is scheduled. These filtered results are also forwarded to the **materializer** module.
At the **materializer**, the dynamic query results are filled into the static query that results into materialized static queries which are then forwarded to the **proxy cache**.
The **result manager** will look up previous results of those materialized queries in its **cache**, or otherwise execute them against the TPF client. Results for each of those materialized queries are then sent to the client.

Note that the principle of caching static queries has been ingrained into the architecture. If this caching is not required because for example all possible queries are fully dynamic in nature, several modules in this process could be skipped. The **rewriter** could simply

[1] Currently active in this context means that the triple has a time annotation which indicates that is is valid for the current timestamp.

forward any incoming query as a dynamic query and returning no static query, this would result in the full query execution by the **streamer** and this would make it possible to completely skip the **materializer**, **result manager** and **cache** modules.

Figure 8.1 shows an overview of this process. Each step in this process will be explained in detail in the following subsections.

The **rewriter** module can be considered as the most complex module in this architecture. The basics of this module will be explained first, together with the basics of the other modules. Section 8.3 will finally discuss some edge cases that are possible regarding the **rewriter** module.

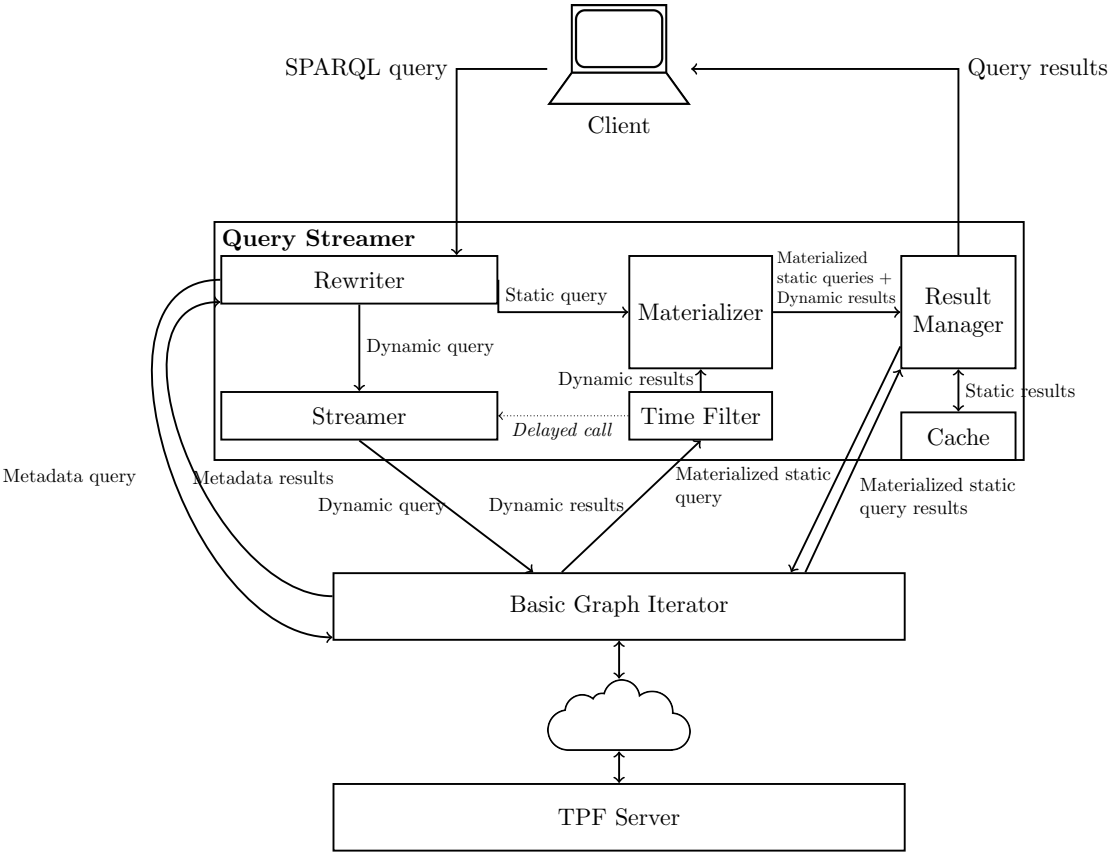


Figure 8.1: Overview of the proposed architecture.

8.2.1 Rewriter

The **rewriter** is responsible for splitting up the original query into a static and dynamic query. This module is only called once for each incoming query, it is part of the preprocessing.

```
PREFIX t: <http://example.org/train#>

SELECT ?delay ?departureTime ?headSign
WHERE {
  _:id t:delay ?delay .
  _:id t:departureTime ?departureTime .
  _:id t:headSign ?headSign .
}
```

Listing 8.1: SPARQL query for requesting the delay, departure occurringtime and head sign for all trains that will be split up into a static and dynamic part.

```
PREFIX t: <http://example.org/train#>

SELECT ?delay ?final
WHERE {
  GRAPH ?stmt {
    _:id t:delay ?delay .
  }
  ?stmt tmp:expiration ?final .
}
```

Listing 8.2: Temporary SPARQL query for the first triple pattern from Listing 8.1 to check if it is dynamic.

To illustrate the workings of this module, the query from Listing 8.1 will be used as input to this module. The examples in this chapter will always use graph-based time-annotation based on expiration times. For the sake of the example, we will assume that the first triple pattern of this query is dynamic, and the two following patterns are static. The pattern being dynamic means that it is time-annotated in some data store.

An important requirement for this module, is that it must be able to distinguish static from dynamic triple patterns. One way of doing this is by making a new temporary query for every triple pattern and assume it is time annotated. Each temporary query is then executed against the TPF client. If this query results in at least one match, then this triple pattern is assumed to be dynamic. Otherwise, if no results were found, the triple pattern does not have a time annotation and is assumed to be static. Listing 8.1 shows an example of such a query for the first triple pattern from the query in Listing 8.1. Since this triple pattern truly is time annotated, this query will have a non-empty result, and the algorithm will mark this pattern as dynamic.

Another step before the actual query splitting is the conversion of blank nodes to variables. To understand why this step is required, one has to look at the end result of the query splitter. 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 inside 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, the blank nodes have to be converted to variables, while maintaining their semantics. This means that *a*) two identical blank nodes occurring in different triple patterns must eventually

```
PREFIX t: <http://example.org/train#>

SELECT ?id ?departureTime ?headSign
WHERE {
  ?id t:departureTime ?departureTime .
  ?id t:headSign ?headSign .
}
```

Listing 8.3: Static SPARQL query created from the query in Listing 8.1.

```
PREFIX t: <http://example.org/train#>

SELECT ?id ?delay
WHERE {
    GRAPH ?stmt {
      ?id t:delay ?delay .
    }
    ?stmt tmp:expiration ?final .
}
```

Listing 8.4: Dynamic SPARQL query created from the query in Listing 8.1.

have the same variable. And *b*) newly introduced variables must not mutually collide and must not collide with existing variables.

The rewriter will iterate over all the triple patterns in the original query. Each triple pattern is then added to either the static or dynamic query depending on the result from its temporary query, while maintaining the hierarchical location of the pattern inside the query. This means that if for example triple pattern P is part of a graph G and P is dynamic, that P should also be placed under graph G inside the dynamic query. All patterns that are added to the dynamic query are added in their time annotated form.

Once the triple patterns have been split up into static and dynamic queries, only one more step is required to result in a real static and dynamic query. The graphs of the static and dynamic parts can either be *connected* or *disconnected*. In case they are disconnected, the two parts are in fact two separate queries executed as one, so no merging of the end results of the queries is required. In case the graphs are connected, the end results of the static and dynamic queries have to be merged. This merging can only be done if we know the variables that connect these two graphs. It is possible (and common) for a SPARQL query to use variables inside the `WHERE` clause that are not present in the `SELECT` clause. Since we want to join these static and dynamic results later on, we must know the values of these variables. This can be solved by making sure that all variables from the `WHERE` clause are guaranteed to be present inside the `SELECT` clause. To prevent the client from ever seeing these additional variables, the end results go through a variable filter that will only let through the variables that were originally requested by the client.

The example query from Listing 8.1 will after splitting look like the static and dynamic query respectively found in Listings 8.3 and 8.4.

After the splitting process, the static query is forwarded for later usage to the **material-**

izer, which is further explained in Subsection 8.2.4. The dynamic query is forwarded to the **streamer** module which is then immediately triggered to produce a first streaming result, this is explained hereafter.

8.2.2 Streamer

The streamer module is the starting point of the circular calls of the system. It can be called by either the **rewriter**, which initiates the query result stream, or by the **time filter**, which is called when the results from this module come in.

The only thing this module actually does is storing the dynamic query it received from the **rewriter**, and executing that query against the **TPF client**. The results from that query will be received by the **time filter**.

8.2.3 Time Filter

This module will filter all results from the dynamic query that was triggered by the **streamer**. This filtering is based on the time annotations of the query results. The query results were part of a dynamic query, so this implies that all possible results are time-annotated.

Filtering will occur based on the current time and the time annotation per result. In case we have a time interval annotation for a certain result, a check will be done whether the current time lies within this interval, and only then will the result be allowed to pass through the filter. In case we have a simple expiration time for each result, only those results that are not yet expired will be passed through the filter.

All dynamic query results that successfully went though the filter will be forwarded to the **materializer**.

Based on all the filtrates, a minimum expiration time will be determined. If the annotation type is a time interval, the minimum of all interval expirations is taken. Otherwise, if annotation is done with expiration times, the minimum of these times is taken. We then schedule a new call to the **streamer** module at that newly determined minimum time. This will make sure that all dynamic results will be kept up to date.

In future work, some optimizations might be possible in regards to the minimum expiration time determination. The granularity of change might differ significantly between different dynamic triple patterns, so these could be split up into separate queries similar to the process of splitting up static and dynamic queries.

```
PREFIX t: <http://example.org/train#>

SELECT ?departureTime ?headSign
WHERE {
  t:train4815 t:departureTime ?departureTime .
  t:train4815 t:headSign ?headSign .
}
```

Listing 8.5: A materialized static SPARQL query based on the query in Listing 8.3 using the value `<http://example.org/train#train4815>` for the `?id` variable.

8.2.4 Materializer

The **materializer** module is responsible for merging dynamic query results with the static query. This static query was received from the **rewriter** module in the pre-processing phase, this query will never be changed. But each time this materializer is called, a copy of that query is used to modify and forward.

For all variables per dynamic query result, we will look for these variables inside the static query. For each matched variable, we will replace that variable with the materialized value for that variable that is present in the dynamic query result. Note that we might have multiple results from the dynamic query, so this materialization step will produce multiple materialized static queries in that case. Simply put, this step will fill in the dynamic query results into the static query.

Once the materialization step is done, the newly created queries will be forwarded to the **result manager** together with the dynamic query results.

For example, assume we have found exactly one result from the dynamic query found in Listing 8.4, the result being: `{ "?id": "<http://example.org/train#train4815>", "?delay": "\"P10S\"^^xsd:duration" }`. If we fill in this one result into the static query from Listing 8.3, this resulting materialized static query will look like the one in Listing 8.5.

8.2.5 Result Manager

This is the last step in the streaming loop for returning the 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. A requirement for this identifier is that is 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.

```
PREFIX t: <http://example.org/train#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>

SELECT ?id ?departureTime
WHERE {
  ?id t:departureTime ?departureTime .
}
```

Listing 8.6: Static part of a query for retrieving the departure time of all trains.

```
PREFIX tmp: <http://example.org/temporal#>
PREFIX sp: <http://example.org/singletonproperties#>
PREFIX t: <http://example.org/train#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>

SELECT ?id ?delay ?final
WHERE {
  GRAPH ?stmtExpiration {
    ?id t:delay ?delay .
  }
  ?stmtExpiration tmp:expiration ?final .
}
```

Listing 8.7: Dynamic part of a query for retrieving the delay of all trains.

To determine this connection, the intersection of all[2] static and dynamic variables needs to be taken. We can materialize this connection by filling in the variables with the dynamic query results. Because the intersection of static and dynamic variables was taken, and we already have the dynamic results, all variables are guaranteed to be filled in.

This graph connection can easily be converted into a cache identifier by concatenating all pairs of variable-name, variable-result.

In case of a query that calculates the departure time and delay of all trains that is split into a static and dynamic part which can be seen in respectively Listings 8.6 and 8.7, the first step would be to determine all possible variables for the static and dynamic query. In this case it is clear that the `SELECT` clause of the static query already contains all variables from the body, being `?id` and `?departureTime`. The dynamic part however has the variables `?id`, `?delay ?final` and `?stmtExpiration`. Note that the variable `?stmtExpiration` does not appear in the `SELECT` clause of the dynamic query, so this variable has to be added to this clause for ensuring the correct workings of the algorithm. When variables are added to the `SELECT` clause, they will not be delivered to the user as a result from the original query. The intersection of these two lists of variables is {`?id`}. Since `?id` is guaranteed to be a result of the dynamic query, this variable can be used as the link between the two query graphs and can thus be used as an identifier for the caching mechanism. If a value `"train:4815"` for `?id` was found, the identifier could look like: `"?id=train:4815"`.

[2]Not only the variables occurring in the `SELECT` clause, but all variables that occur in the `WHERE` clause. This is because SPARQL allows you to use variables inside the `WHERE` clause to *connect* triples without them needing to be part of the `SELECT` clause.

Now that we have an identifier for each materialized static query, a cache lookup can be done to check if query results are already present. If they are available, those results are merged with the dynamic results and only the variables that were part of the original query's `SELECT` clause will be sent back to the client. Otherwise, if the cache did not contain such a key, that materialized static query will be executed against the **TPF client** and all results will be cached with the identifier as key. Those results are then also sent back to the client in the same manner as in the case where the cache already had the results.

8.3 Rewriter Edge Cases

8.3.1 `DISTINCT` Query Splitting

Since this solution does not return any of the time annotations used internally, it is possible in some special cases that duplicate results appear. When the streaming encounters a large amount of network delay, it could be possible that a long query execution time results in multiple timely results of the same fact. This can occur when a certain dynamic result is received that is valid at that given time, but because of the slow query execution another consecutive value of that fact could become active in the dataset. Since TPF executes queries by retrieving partial results of the dataset, this could result in this newly added fact being retrieved after its predecessor, while still being active at the time of arrival at the client.

The SPARQL protocol has the `DISTINCT` modifier which is used to declare that duplicate query solutions have to be eliminated. So when this modifier is active in the original query, the streamer has to select one of these versions and return them to the user.

A first possible solution to this problem could be to buffer all results in the streaming module, and only returning the latest version to the client. The disadvantage of this is the delay in results, because even though some results might already be waiting in the streamer, the user still has to wait to see them.

An alternative solution is by not returning the latest version, but the first version of the fact. This solves the problem of delay in results, because no buffering is required anymore, but this is at the cost of not having the very latest version of data.

8.3.2 Query Splitting with `UNIONS`

It is not always possible to perfectly split up patterns from `UNIONS` to one static and one dynamic query. For example, the query in Listing 8.8, can not be split up in just one static and dynamic query assuming triples with the `t:delay` predicate are the only ones who are dynamic. This is because the union makes it impossible to have disjunct static and dynamic queries, so additional queries would be required. A naive solution to this problem is that the complete `UNION` can be assumed to be dynamic, this way

```
PREFIX t: <http://example.org/train#>

SELECT ?departureTime
WHERE {
  {
    _:id t:delay "P10S"^^xsd:duration .
  } UNION {
    _:id t:headSign "Gent-Sint-Pieters"^^xsd:string .
  } UNION {
    _:id t:headSign "Brugge"^^xsd:string .
  }
  _:id t:departureTime ?departureTime .

}
```

Listing 8.8: SPARQL query for finding the departure times of all trains either having a delay of ten seconds or departing from Gent-Sint-Pieters or Brugge.

the results are guaranteed to be correct, at the cost of possibly some data transfer overhead caused by static triple patterns not being cached. There are however some optimizations possible for this problem when for example we only have a `UNION` of two graphs, but this is currently not the main goal of this research and must be looked into in a future research.

8.3.3 Query Splitting with Implicit Graphs

As was explained in Subsection 7.2.4, implicit graphs are identified by triples. The rewriter module loops over every triple pattern to check if they are dynamic. When using implicit graph annotation, this means that the rewriter has to check if that triple pattern has a time annotation in its context. Because these are triple patterns with variables or blank nodes, multiple triples could be matched with them, so the rewriter has to check the context of all possible triple pattern bindings to find at least one triple that has a time annotation in its context, and in that case the triple pattern is considered dynamic.

This of course introduces some major computational and data transfer overhead. As future work, it might be possible to build implicit graphs for triple patterns as well, so that the server determines all possible triple patterns of a dynamic triple and adds some signaling triple to all of them saying that this pattern is dynamic.

8.3.4 Dependent Pattern Extraction

Some queries will require certain static triple patterns to also be considered dynamic to ensure the correct working of the materializer module. We will refer to these special static triple patterns as *dependent triple* patterns because they directly depend on certain dynamic patterns. When we for example have the query in Listing 8.9 with the first pattern being a dynamic one, and the second being static. This would normally

```
PREFIX t: <http://example.org/train#>

SELECT ?id ?delay
WHERE {
  ?id t:delay ?delay .
  ?id t:headSign "Gent-Sint-Pieters" .
}
```

Listing 8.9: SPARQL query with a static and dynamic triple pattern where the static pattern needs to be considered dynamic.

result in two queries, one for the dynamic pattern and one for the static pattern. This dynamic query would then be executed and several results for the variables `?id` and `?delay` might be found. According to the algorithm, these results are then used to build materialized static queries, and this is where the algorithm gets in trouble. As you might have noticed, the `?id` variable will be in the intersection of the variables used for both the static and dynamic query, which will be called the *intersected variables* from now on. This means that this variable simply has to be filled into the static query for materialization. But when this is done, there are no more variables in the static query, making a query request impossible.

A solution to this problem could be to add an additional step right after the query splitting step that is responsible for looping through all triple patterns in the static query and moving them to the dynamic query if the difference of their variables with the intersected variables results in an empty set. These extracted triple patterns are the *dependent triple patterns*.

Chapter 9

Use Case

This chapter will first discuss several derived data models of the use case regarding SPARQL queries on train departure information based on the *basic data model* introduced in Chapter 3. The use case from that chapter will be used to perform measurements on the implementation of our solution and to compare it with alternatives like C-SPARQL [55] and CQELS [10].

An implementation of the architecture explained in Chapter 8 is created in JavaScript using Node.js [64], just like the original Triple Pattern Fragments client on which this solution is based. This implementation has support for all temporal domains and types of time annotation explained in Chapter 7.

The query from Listing 9.1 is used to retrieve information on all departures stored in the Triple Pattern Fragments server defined in the *basic data model* as was shown in Figure 3.1. This query will be referred to as the *basic query*, as it will be altered inside the rewriter depending on the annotation type and temporal domain. Our implementation will then provide a stream of query results to the user, updating each time new results become available or existing results are changed.

```
PREFIX t: <http://example.org/train#>

SELECT DISTINCT ?delay ?headSign ?routeLabel ?platform
    ?departureTime
WHERE {
    _:id t:delay ?delay .
    _:id t:headSign ?headSign .
    _:id t:routeLabel ?routeLabel .
    _:id t:platform ?platform .
    _:id t:departureTime ?departureTime .
}
```

Listing 9.1: The basic SPARQL query for retrieving all train departure information.

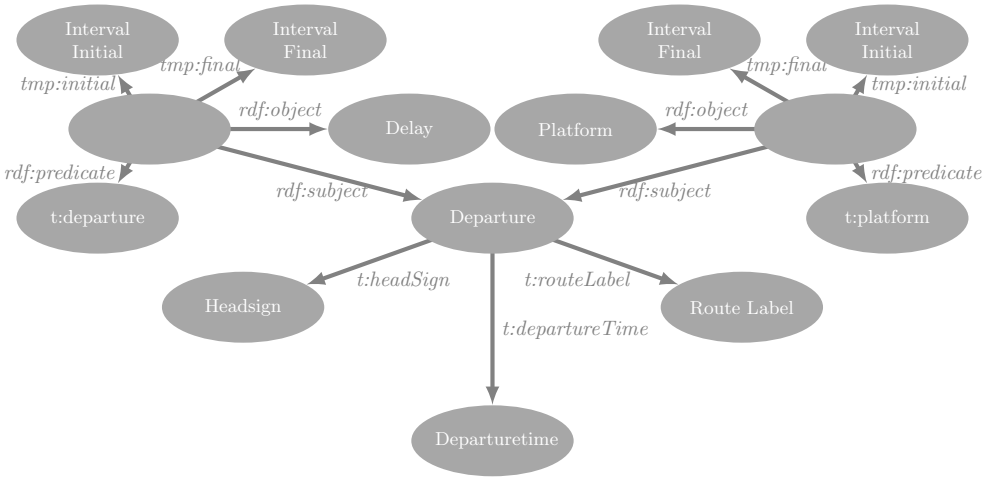


Figure 9.1: Reification data model for representing train departures in one train station using time intervals.

9.1 Derived Data Models

This section will show the different possible data models derived from the basic data model for each type of time annotation, for both the time interval and expiration time domain.

The nodes `Interval Initial` and `Interval Final` respectively refer to the start and end time of the time interval annotation, both with data type `xsd:dateTime`. While the node `Expiration` refers to the expiration time annotation, with data type `xsd:dateTime`.

The time interval-based annotation for the four annotation types (as discussed in Chapter 7) can be found in Figures 9.1, 9.2, 9.3 and 9.4.

There aren't any real dynamic triples anymore in these data models as was the case with the basic data model. The dynamic triples received time annotations and have become static in their context.

The expiration time-based data models are completely analogous, with the difference that the nodes `Interval Initial` and `Interval Final` would be merged to one node `Expiration` and the joined edge would be named `tmp:expiration`.

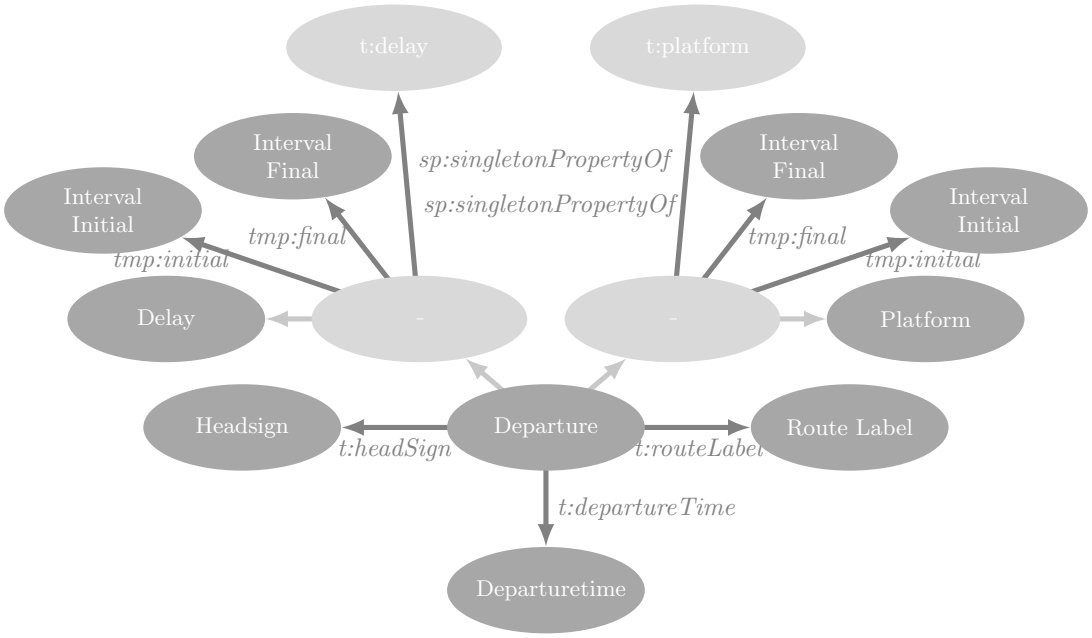


Figure 9.2: Singleton properties data model for representing train departures in one train station using time intervals. The nodes in light grey are predicates.

9.2 Measurements

9.2.1 Annotation Types

Our use case will be executed for all eight possible annotation combinations in our implementation. As mentioned before, the timeliness of the server data is ten seconds. Each test will be executed for a duration of one minute on the same dataset that was recorded from the iRail API. Each possible test will be run ten times and the final results will be taken as the average of these.

A naive implementation was created using the basic query from Listing 9.1. This naive test is run on the default TPF client by continuously polling with this basic query at several different frequencies.

9.2.2 Server and Client Performance

We will also compare the performance of our solution in terms of processor efficiency with C-SPARQL and CQELS. The reason for these two is that they respectively do black box and white box query processing, which will most likely produce different results. The setup for each of these tests will consist of one server which has an endpoint that is either TPF, C-SPARQL or CQELS, depending on which test is being executed. This

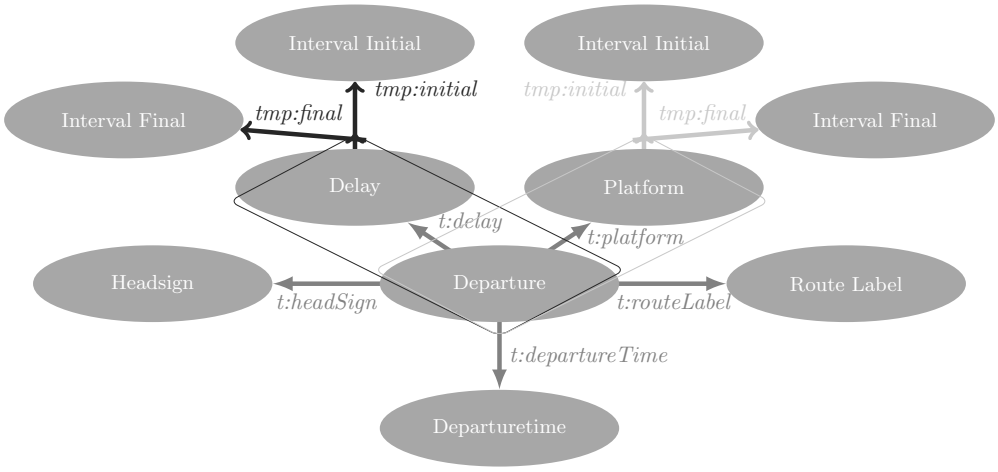


Figure 9.3: Graphs data model for representing train departures in one train station using time intervals. The surrounded facts indicate graphs which can be refered to as a node. For clarity, the edges leaving from graphs are indicated in their greyscale color.

server can be reached by ten different clients which each have a increasing amount of concurrent query executions. These clients can query using our query streamer implementation, C-SPARQL or CQELS, depending on which test is currently being executed. Initially, each client will execute just one query which will stream for the duration of one minute. These ten clients will be synchronized so that they start each query execution at the exact same moment. After that stream, each client will initiate two simultaneous query streams. This will be repeated until each client has executed 20 simultaneous streams. From the point of view of the server this results in a series of 10 to 200 concurrent query streams, incremented by 10 each minute. An overview of this setup can be found in Figure 9.5.

A dynamic query generator will make sure that each client query execution has a different query, these generated queries are equivalent for the different streaming approaches. This dynamic query generator will derive queries from the basic query in Listing 9.1 when testing our implementation. For C-SPARQL and CQELS, equivalent basic queries have been created which can respectively be found in Listing 9.2 and Listing 9.3. Note that for C-SPARQL the stream `http://example.org/stream` contains the dynamic data for both the delay and platform, while for CQELS these respectively exist within the streams `http://example.org/streamdelay` and `http://example.org/streamplatform`. For C-SPARQL, the graph `http://example.org/static` contains our static data while for CQELS this data exists within the default graph.

9.2.3 Setup Details

All of these tests ran on the Virtual Wall (generation 2) [65] environment from iMinds. This allowed us to easily define the required machines with their specifications. Most notably the scalability test setup from Figure 9.5 was very convenient to setup in this

```
REGISTER QUERY TrainDepartures AS
PREFIX t: <http://example.org/train#>
SELECT ?delay ?headSign ?routeLabel ?platform ?departureTime
FROM STREAM <http://example.org/stream> [RANGE 1h STEP 10s]
FROM <http://example.org/static>
WHERE {
  _:id t:delay ?delay .
  _:id t:headSign ?headSign .
  _:id t:routeLabel ?routeLabel .
  _:id t:platform ?platform .
  _:id t:departureTime ?departureTime .
}
```

Listing 9.2: The C-SPARQL query for retrieving all train departure information. This is equivalent to the basic query in Listing 9.1.

```
PREFIX t: <http://example.org/train#>
SELECT ?delay ?headSign ?routeLabel ?platform ?departureTime
WHERE {
  STREAM <http://example.org/streamdelay> [RANGE 60m SLIDE 10s] {
    ?id t:delay ?delay
  }
  STREAM <http://example.org/streamplatform> [RANGE 60m SLIDE 10s] {
    ?id t:platform ?platform
  }
  ?id t:headSign ?headSign .
  ?id t:routeLabel ?routeLabel .
  ?id t:departureTime ?departureTime .
}
```

Listing 9.3: The CQELS query for retrieving all train departure information. This is equivalent to the basic query in Listing 9.1.

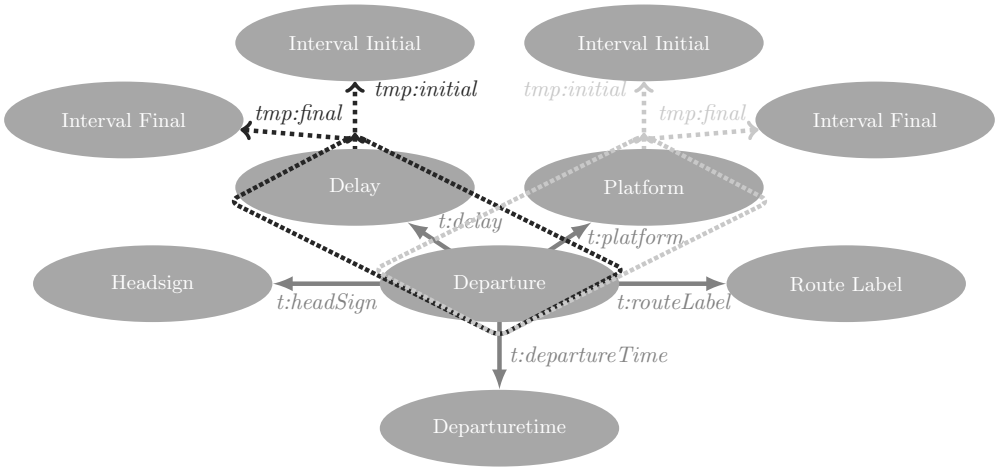


Figure 9.4: Implicit graphs data model for representing train departures in one train station using time intervals. The surrounded facts indicate implicit graphs which can be refered to as a node. For clarity, the edges leaving from graphs are indicated in their greyscale color.

environment. Each machine used had a two Hexacore Intel E5645 (2.4GHz) CPU's with 24GB RAM and was running Ubuntu 12.04 LTS. Our scalability tests used the CQELS engine version 1.0.1 [66] and the C-SPARQL engine version 0.9 [67].

In each of our tests, the TPF endpoint had access to around 300 static triples at the start of the experiment, with around 200 dynamic triples that were created and removed each ten seconds. To approximate real-world conditions, each test used a deterministically defined delay for each HTTP request. This delay is randomly chosen from a Poisson distribution with a fixed seed resulting in delays around 100 milliseconds. The reason for using the same seed for each test is to make sure that for example the *nth* HTTP request using reification has the exact same delay as the *nth* request using graphs.

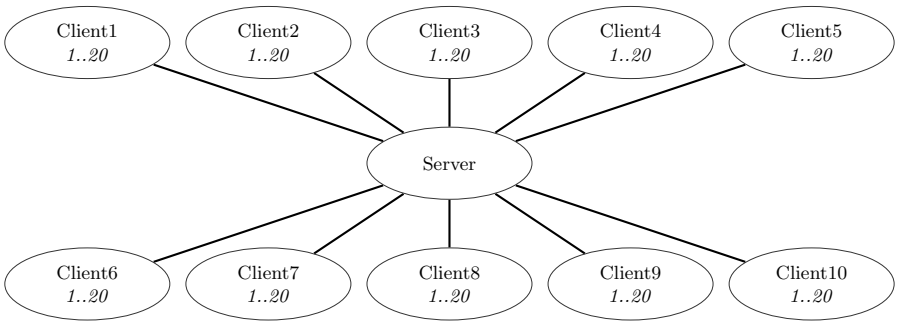


Figure 9.5: Setup for the server and client performance tests. Each client will execute 1 to 20 simultaneous queries each sixty seconds. This setup exists for our implementation, C-SPARQL and CQELS.

Chapter 10

Results

Several tests on our implementation were executed for the use case explained in Chapter 8. First, we will compare the different dynamic data representation types of our implementation. After that, our implementation will be compared with a naive implementation. Finally, a performance comparison in terms of processor efficiency will be made with alternative streaming SPARQL approaches like C-SPARQL and CQELS.

10.1 Dynamic Data Representation Types

This section will compare the performance of each possible dynamic data representation type using the test results from Figure 10.1. Each subfigure contains subsequent requests inside one streaming query execution for the four types of time annotation types. The first row of subfigures shows the annotations using time intervals, while the second row shows the expiration times. The first column of subfigures has caching disabled and the second column has caching enabled. The following subsections will discuss several aspects of these figures.

10.1.1 Temporal Domains for Timeliness

The difference between time interval annotation and annotation using expiration times, respectively the first and second row of subfigures in Figure 10.1, now becomes very apparent. In Subsection 7.1.2 we discussed the main differences between these two types of temporal domains. It was also mentioned that time interval annotation causes growing datasets when expired fact versions are not removed, which results in continuously slower query execution times. This becomes very clear in Figure 10.1, the time interval annotation execution times continuously increase, while the expiration execution times stay more or less constant.

When we compare the difference in data transfer between the time interval and expiration time approaches in Figure 10.2, we can indeed confirm that the increasing query times are in fact caused by the larger data set, which is the cause of this increased data transfer.

Annotation using time intervals can still be useful. In some scenarios one could need multiple versions of the same fact where the single latest version is not sufficient, what can only be achieved using time intervals. In that case it is very important to implement an aggregation or archival process for old versions of these facts. The rate at which these processes should occur are determined by the timeliness of these facts. For example in this use case it could be useful to only keep the three latest versions of each fact, what would result in a constant amortized execution time.

10.1.2 Methods for Time Annotation

This section will discuss the four different annotation methods, these are the four plots inside each subfigure of Figure 10.1. It is clear that the reification approach has in all cases the worst performance. This is a direct result from the large amount of triples this approach requires for each fact, which leads to higher execution times. The three other approaches have execution times which are significantly lower than those of reification, but there still is a clear distinction.

Figure 10.3 contains the same data as Figures 10.1c and 10.1d, but at a lower scale to more clearly see the difference between the performance of the three remaining approaches. Figures 10.1 and 10.3 show that there is a clear order in the performance of the four methods of annotation. The graph approach has the overall lowest execution times, closely followed by first the singleton properties approach and then implicit graphs, finally the reification approach has the highest overall execution times.

This order of performance closely resembles the triple-count functions from Table 7.1, which indicates that the largest part of the execution times is determined by the amount of triples that is required by the annotation type, which is in line with our explanation of the slower execution times with time interval annotation. This is also confirmed when looking at the data transfer plots for the four annotation methods in Figure 10.4.

When we look at the triple-count functions from Table 7.1, we see that both graph approaches require the same amount of triples, but this does not result in the same performance. These increased execution times of the implicit graph approach can be explained by the extra step of indirection that is required to determine the (implicit) graph of its dynamic triples. This extra step of indirection always requires the execution of an extra query, as explained in Subsection 7.2.4.

10.1.3 Cache Influence

Chapter 8 explained the architecture of our solution in which a caching mechanism was available, which is responsible for remembering results from the static queries. We also explained that this caching mechanism could be disabled, and this subsection explains what the influence of this caching is on the performance of the execution.

The first columns of Figures 10.1 and 10.3 show execution times without caching, while the second columns show the same with caching enabled. In (almost) all cases, this cache has a very positive effect on the execution times: the average execution time is almost halved. And it can be seen that when caching is enabled that the first execution time per stream is higher than the following ones. With time intervals this effect is not clear anymore because of the increasing execution times. This means that our cache does what it's supposed to do, it accumulates more data the first execution time which can be reused later on.

Only in the case of reification, this cache seems to have a bad effect on the performance, which causes the incompleteness of test results for this case. To understand what can cause this behavior, we have to look back at how this caching mechanism works. Subsection 8.2.5 explained that the result manager module handles caching by saving results of the static query. Assume we are using time intervals in our use case, for reification this would result in the query from Listing 10.1 when caching is disabled while Listings 10.2 and 10.3 respectively contain the static and dynamic query when caching is enabled. The execution plan for SPARQL queries by the TPF client always makes sure the most informative triple patterns in the given query are queried before the less informative ones. When we are not using the caching mechanism, the query from Listing 10.1 will result in one of the last three triple patterns to be considered as the most informative one, because the other patterns will have exactly double the amount of results than these three when using the data model from Figure 9.1. When caching is now enabled, the dynamic query from Listing 10.3 must be separately executed before the static query from Listing 10.2, but our previously most informative triple patterns now can't be selected anymore, since they are part of the static query that will be executed later on. This results in at least double the execution time, which in our case causes the execution to be too long to fully fit into our test case.

When we have another look at out basic use case query in Listing 9.1, we see that there are two dynamic triples and three static triples. In theoretically optimal caching scenarios, this could lead to an execution reduction of 60%. In our test results, the caching leads to an average reduction of 56%[1], which is slightly lower than the optimal case. The main reason for this lower reduction is the fact that the dynamic query simply consumes a relatively larger portion of the execution time. This is because of the overhead that is introduced when using annotations.

The plots in Figure 10.5 show us a very interesting effect of the cache on the data transfer. The cache reduces the amount of data transfer as expected, but it also increases the overall throughput what results in an even faster query execution in the stream.

[1]This average was taken without the results from reification, since these distorted the average too much.

```
PREFIX tmp: <http://example.org/temporal#>
PREFIX sp: <http://example.org/singletonproperties#>
PREFIX t: <http://example.org/train#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>

SELECT ?id0 ?platform ?initial0 ?final0 ?delay ?initial1 ?final1 ?
    departureTime
    ?headSign ?routeLabel
WHERE {
  _:stmt5 rdf:subject ?id0 .
  _:stmt5 rdf:predicate t:platform .
  _:stmt5 rdf:object ?platform .
  _:stmt5 tmp:intervalInitial ?initial0 .
  _:stmt5 tmp:intervalFinal ?final0 .
  _:stmt6 rdf:subject ?id0 .
  _:stmt6 rdf:predicate t:delay .
  _:stmt6 rdf:object ?delay .
  _:stmt6 tmp:intervalInitial ?initial1 .
  _:stmt6 tmp:intervalFinal ?final1 .
  ?id0 t:departureTime ?departureTime .
  ?id0 t:headSign ?headSign .
  ?id0 t:routeLabel ?routeLabel .
}
```

Listing 10.1: Time-annotated SPARQL query for our use case using the reification approach with time intervals without caching.

```
PREFIX t: <http://example.org/train#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>

SELECT ?id0 ?headSign ?routeLabel ?departureTime
WHERE {
  ?id0 t:headSign ?headSign .
  ?id0 t:routeLabel ?routeLabel .
  ?id0 t:departureTime ?departureTime .
}
```

Listing 10.2: Static time-annotated SPARQL query for our use case using the reification approach with time intervals with caching.

```
PREFIX tmp: <http://example.org/temporal#>
PREFIX sp: <http://example.org/singletonproperties#>
PREFIX t: <http://example.org/train#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>

SELECT ?id0 ?delay ?initial0 ?final0 ?platform ?initial1 ?final1
WHERE {
  _:stmt5 rdf:subject ?id0 .
  _:stmt5 rdf:predicate t:delay .
  _:stmt5 rdf:object ?delay .
  _:stmt5 tmp:intervalInitial ?initial0 .
  _:stmt5 tmp:intervalFinal ?final0 .
  _:stmt6 rdf:subject ?id0 .
  _:stmt6 rdf:predicate t:platform .
  _:stmt6 rdf:object ?platform .
  _:stmt6 tmp:intervalInitial ?initial1 .
  _:stmt6 tmp:intervalFinal ?final1 .
}
```

Listing 10.3: Dynamic time-annotated SPARQL query for our use case using the reification approach with time intervals with caching.

This increased throughput with caching is caused by the dynamic query being able to more specifically look for results, while without caching the non-split query would be larger and requires more processing client-side.

10.1.4 Rewriting Phase

The rewriter module has a non-negligible execution time as can be seen when comparing Figures 10.6 and 10.1, but this is still significantly lower than the average query execution time in a query stream.

The most apparent conclusion from this figure is that the rewriting step using implicit graphs is significantly slower than the other three approaches. This immediately follows from the issue with implicit graphs that was explained in Subsection 8.3.3. We are using triple patterns, but we already have to determine the implicit graph if we want to detect whether it is a static or dynamic triple pattern. So all possible triples for the triple patterns have to be queried, which results in this higher execution time.

The three other annotation methods have an equivalent performance compared to their regular query execution. Interval-based annotation here has an overall higher execution time, this is because of the fact that one more triple is required to store a fact version when compared to expiration times. Caching has no significant influence on the rewriting phase in terms of execution time.

The rewriting phase execution time takes about 15% of the average reification execution time, 31% for the singleton properties and explicit graph approaches, and 155% for the implicit graph approach. Even though this contribution is very high in case of implicit graphs, it should be kept in mind that this rewriting phase only occurs once at the start of the stream, so for a long running query the execution time of this phase becomes less

important.

10.2 Naive Performance

Figure 10.7 shows the execution times of the naive implementation for different query frequencies compared to the optimal case of our solution.

In the optimal case where the naive solution has a frequency of ten seconds, we see that it performs about seven times better than our solution. This is because of the extra complexity that is introduced because we are working with extra time annotations inside our data and its processing. Our solution will perform better in scenarios where a larger part of the data is static and can be cached.

It can also be noted that changing the frequency to more than ten seconds will have no significant influence on naive performance anymore.

The reason why our solution performs better when compared to a naive solution with a high frequency, is that this naive solution has to re-execute the full query each time. Our solution can reuse static information about the query that was acquired during previous requests.

In many realistic scenarios, the timeliness of the dynamic data is unknown to the client. A naive implementation would then require an update frequency that is possibly much higher than the actual data timeliness since it has no way of knowing when the data is actually going to change. Figure 10.7 shows that our solution outperforms the naive implementation once the naive frequency is around half a second. This means that in this scenario when a naive polling frequency is used that is twenty times higher than the dynamic data timeliness, it is more efficient to use our solution with graph-based annotation using expiration times with caching enabled. Keep in mind that such a high naive polling frequency would result in a very high server load, this is further explained in the next section. As already noted before, when more static data related to the query can be cached, the more efficient this solution becomes.

10.3 Server and Client Performance

Figure 10.8 shows the comparison of the average server CPU usage for an increasing amount of concurrent clients for C-SPARQL, CQELS and the query streamer implementation as presented in this work. From this plot it can be concluded that our solution is many orders of magnitude less computationally intensive for this use case than C-SPARQL and CQELS, this allows for more simultaneous requests to be handled by our endpoint. The graph of our implementation seems to stay constant when compared to the slope of the graphs of C-SPARQL and CQELS. This increase in server scalability comes however at the cost of an increased amount of bandwidth usage and a computational increase on each client which can be seen in Figure 10.9. Our solution

provides a higher computational scalability at the cost of a lower bandwidth scalability. Note that this increase in bandwidth is already partially solved by the caching of static results, but could be further improved with the use of HTTP caches.

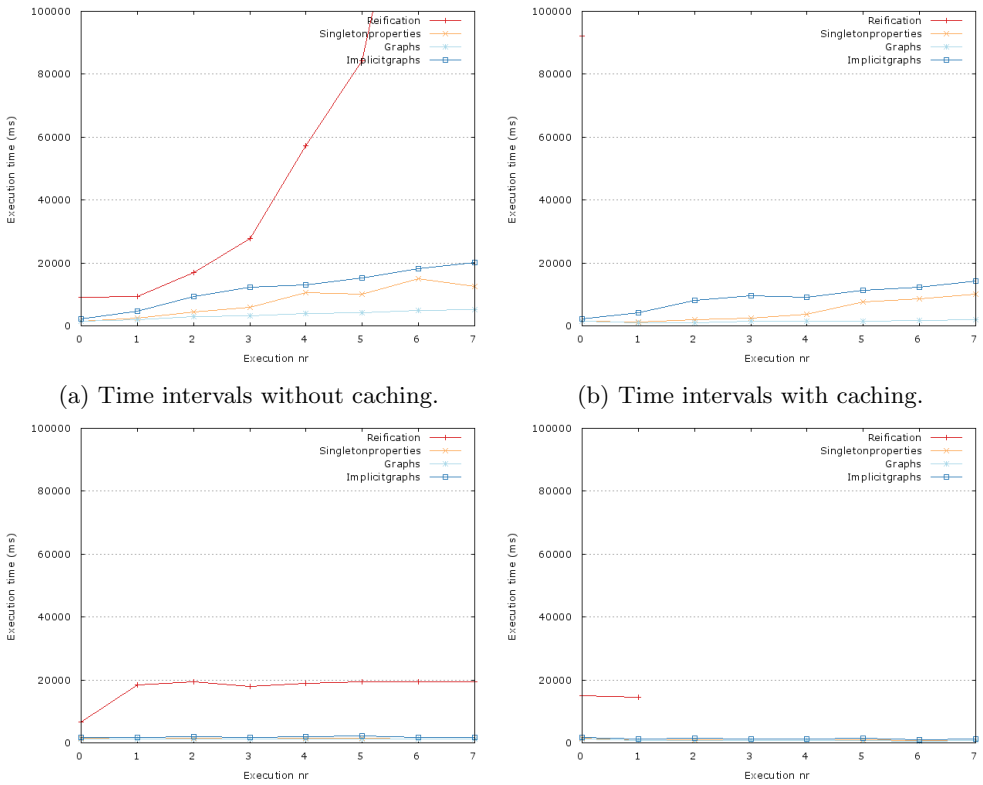
Even though a test like this with 1000 to 10000 simultaneous clients would be more conclusive, these results already give a very clear comparison between two major SPARQL streaming approaches and the implementation presented in this work.

Figure 10.9 and Figure 10.10 respectively show the overall average client CPU usage for the duration of one query stream and the average client CPU usage for a different number of concurrent clients on the same machine. Our solution requires more resources on the client side, this is because of the fundamental idea of TPF that the query calculations should be limited at the endpoint because of which the client must do more work, while C-SPARQL and CQELS require the clients to simply wait for results. Figure 10.9 also shows an initial spike in CPU usage when the query is initialized. This can be explained by the rewriting phase which is executed only once at the start of the query stream together with the initialization of all other required components of this solution. Figure 10.10 shows that it should be avoided to have too many concurrent query streams on the same client.

The plot in Figure 10.8 still doesn't show the complete picture, since each line is an average of many CPU measurements during the test execution which ran for one minute. In order to reveal more detail, Figure 10.11 shows a boxplot of the CPU usage for 200 concurrent clients. There are sixty data points per boxplot with each one indicating the average CPU usage during one second. The location of the whiskers of the boxplot for C-SPARQL show that this approach has very large CPU spikes, which is also confirmed by the many outliers. While our approach has a very low overall CPU usage across all data points. This story is the same for CQELS, but here the average CPU is significantly higher which is explained by the relatively lower performance of CQELS when the dataset size remains relatively small. This unpredictable high load is also true for regular SPARQL endpoints, which has such a problematic influence on their availability. Our approach solves this unpredictability which makes this a sufficient solution to the problem statement of this work.

Figures 10.8 and Figure 10.9 also show that the cost of calculating query results is moved from the server to client, which is one of the main goals of TPF. Because of this, the client has to pay the largest part of the *cost* of calculating query results which directly leads to less server load. This again leads to a higher server availability as was one of the main goals of this research.

Note that these server CPU usage results are not the same as from the original Triple Pattern Fragments measurements [19], there the processor usage reaches more than 20% for 100 clients, while in this case this percentage is between zero and one percent. This is because the original tests execute queries at a high frequency per client, while the clients in our tests are limited to streaming queries with a frequency of ten seconds as defined by the server. The absolute CPU percentages are not the important result, but the relations between these different types and the relative slope for an increasing amount of clients are.



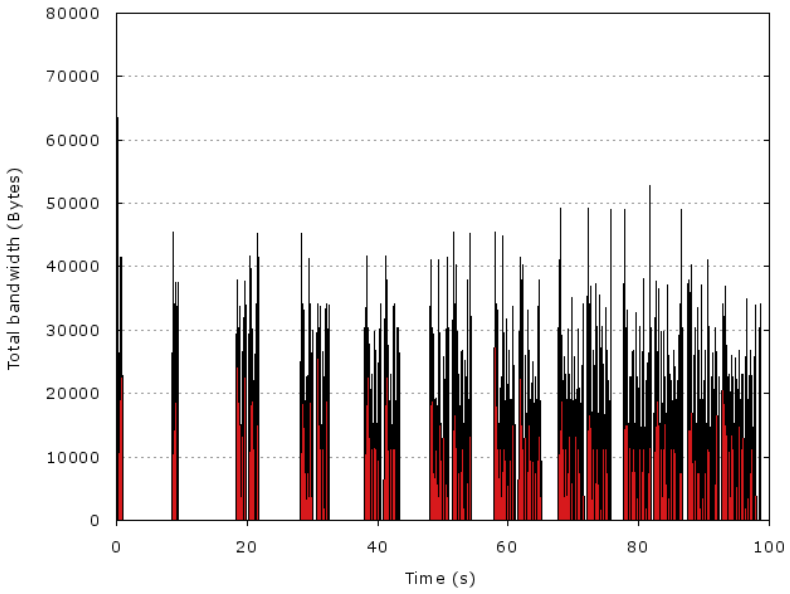
(a) Time intervals without caching.

(b) Time intervals with caching.

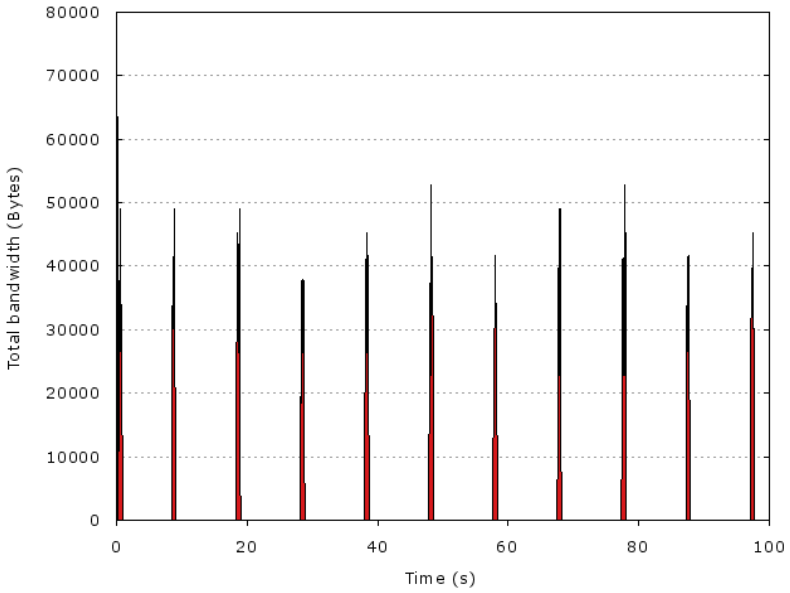
(c) Expiration times without caching.

(d) Expiration times with caching.

Figure 10.1: Executions times for all different types of dynamic data representation for several subsequent streaming requests. The figures show a mostly simultaneous increase when using time intervals and constant execution times for annotation using expiration times.

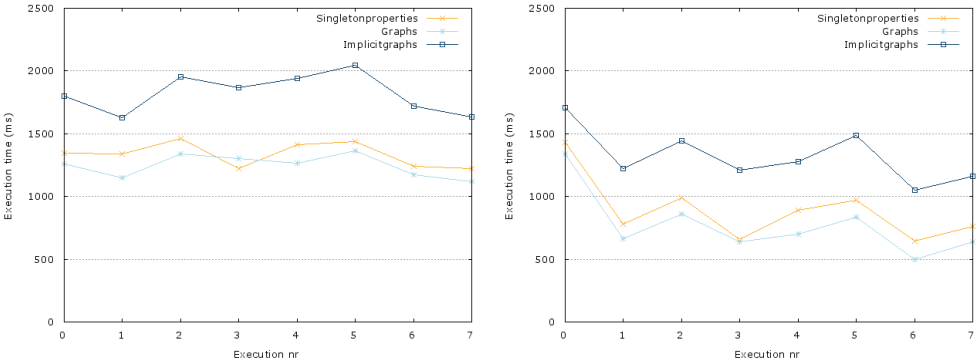


(a) The data transfer in bytes using **time intervals** having a total transfer of 18.28 MB over 4944 requests in this time range.



(b) The data transfer in bytes using **expiration times** having a total transfer of 4.18 MB over 1115 requests in this time range.

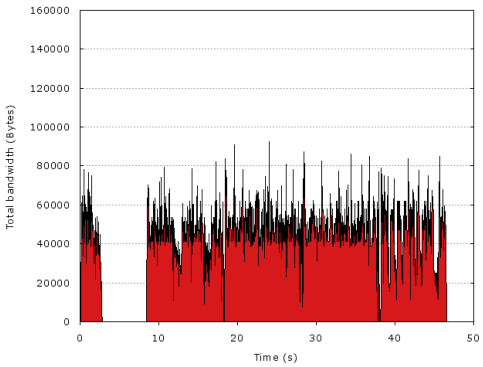
Figure 10.2: The data transfer in bytes for a duration of 100 seconds. These plots used the graph approach with caching disabled. The figures indicate that the time intervals method uses much more bandwidth for the same required information.



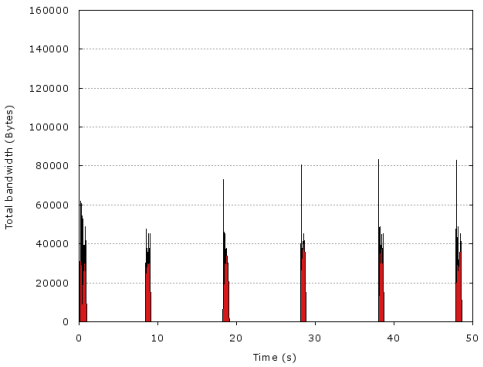
(a) Expiration times without caching.

(b) Expiration times with caching.

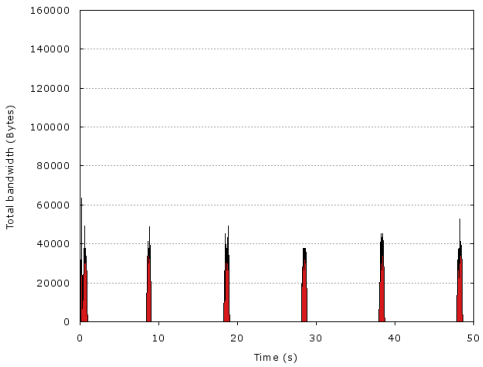
Figure 10.3: Executions times for all different types of time annotation methods using expiration times for several subsequent streaming requests. These figures contain the same data as Figures 10.1c and 10.1d, but without the rapidly increasing reification results in order to reveal the other methods in more detail. They indicate the graph approach having the lowest execution times.



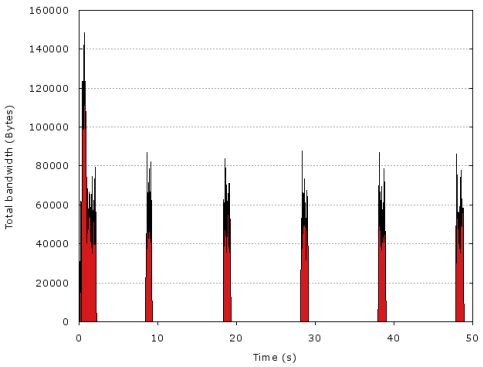
(a) The data transfer in bytes using **reification** having a total transfer of 36.46 MB over 7924 requests in this time range.



(b) The data transfer in bytes using **singleton properties** having a total transfer of 2.57 MB over 627 requests in this time range.

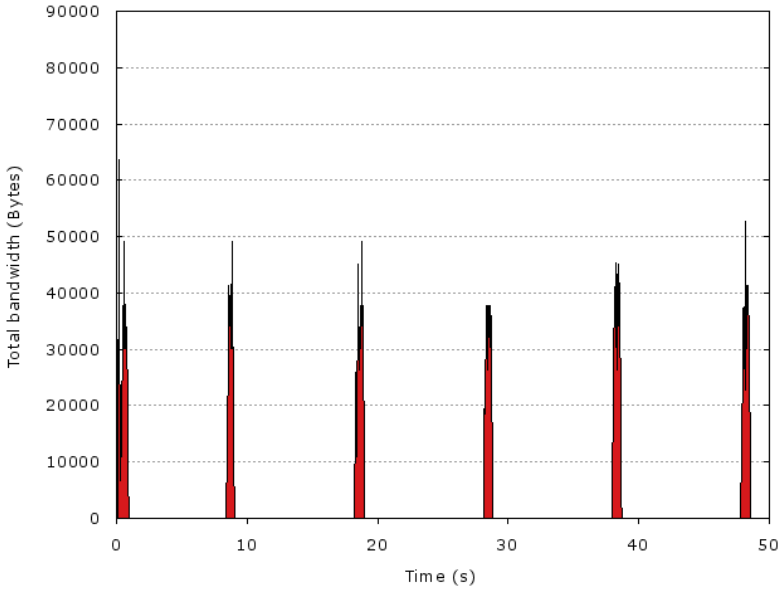


(c) The data transfer in bytes using **explicit graphs** having a total transfer of 2.10 MB over 557 requests in this time range.

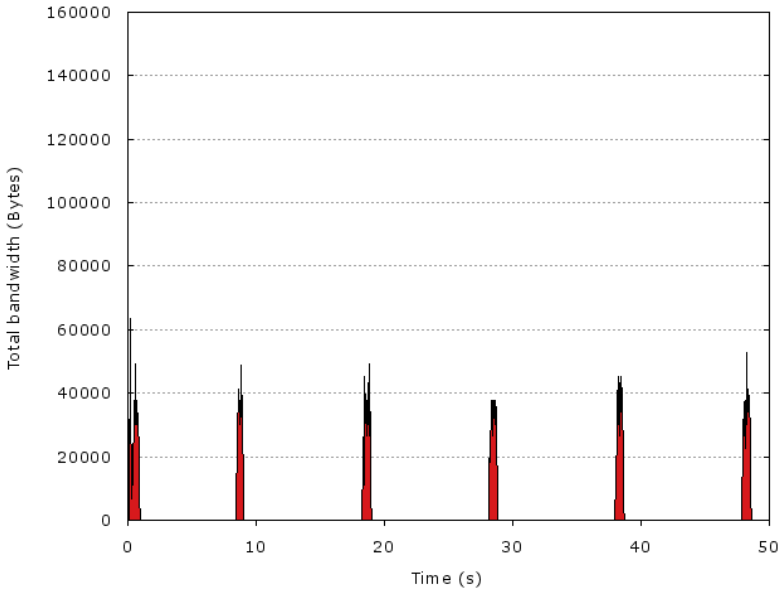


(d) The data transfer in bytes using **implicit graphs** having a total transfer of 6.70 MB over 1191 requests in this time range.

Figure 10.4: The data transfer in bytes for the four annotation methods for a duration of 50 seconds. These plots used expiration times with caching disabled. Reification uses by far the most bandwidth, while the graph approach uses the least.



(a) The data transfer in bytes **without caching** having a total transfer of 2.09 MB over 557 requests in this time range.



(b) The data transfer in bytes **with caching** having a total transfer of 1.75MB over 351 requests in this time range.

Figure 10.5: The data transfer in bytes for a duration of 50 seconds. These plots used the graph approach with expiration times. With caching enabled, the throughput of data is higher while transferring less data in total.

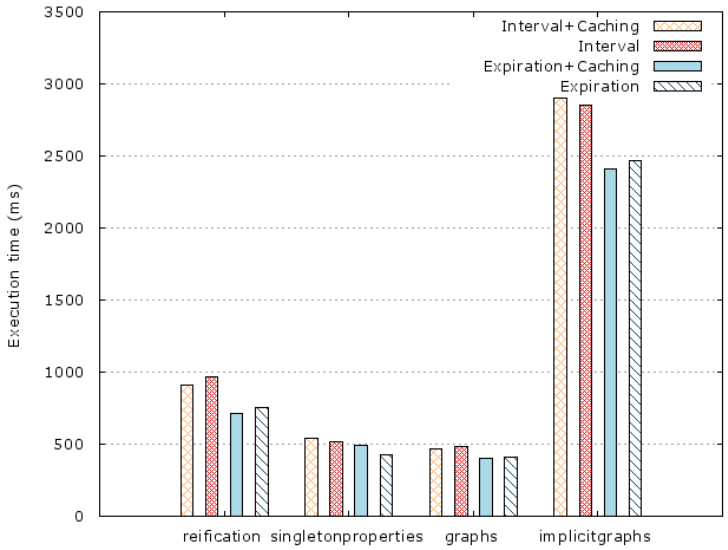


Figure 10.6: Histogram of the preprocessing execution times for the different options for annotation, grouped by annotation method. The graph approach has the lowest preprocessing execution times. Expiration times are slightly faster and caching has no significant influence.

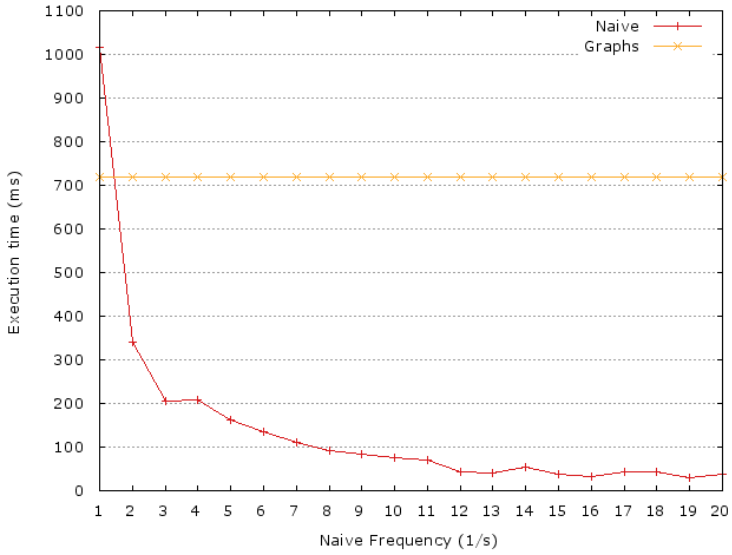


Figure 10.7: Performance of naive implementation compared to the graph approach with expiration times and caching, for different query frequencies in this naive implementation. When the naive frequency is below half a second the graph annotation approach becomes faster.

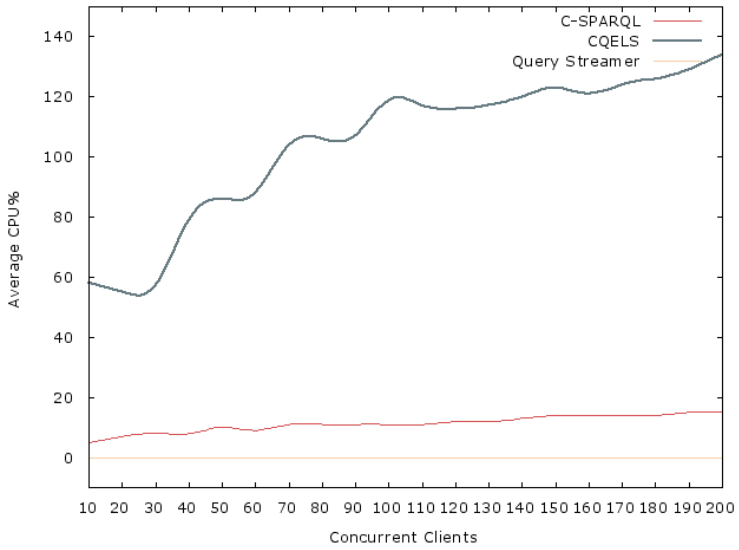


Figure 10.8: Average server CPU usage for an increasing amount of clients for C-SPARQL, CQELS and the solution presented in this work. The CPU usage of this solution proves to be influenced less by the number of clients. Note that the test machine had 4 assigned cores.

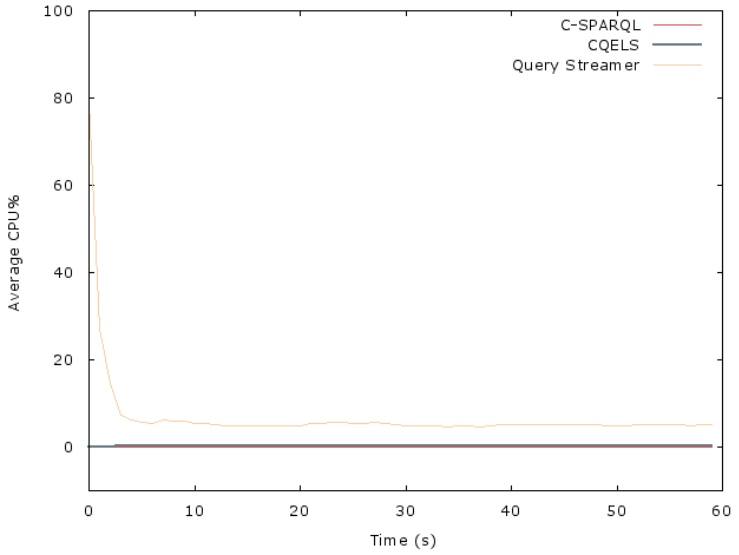


Figure 10.9: Average client CPU usage for one query stream for C-SPARQL, CQELS and the solution presented in this work. Initially the CPU usage for our implementation is very high after which is converges to about 5%. The usage for C-SPARQL and CQELS is almost non-existing.

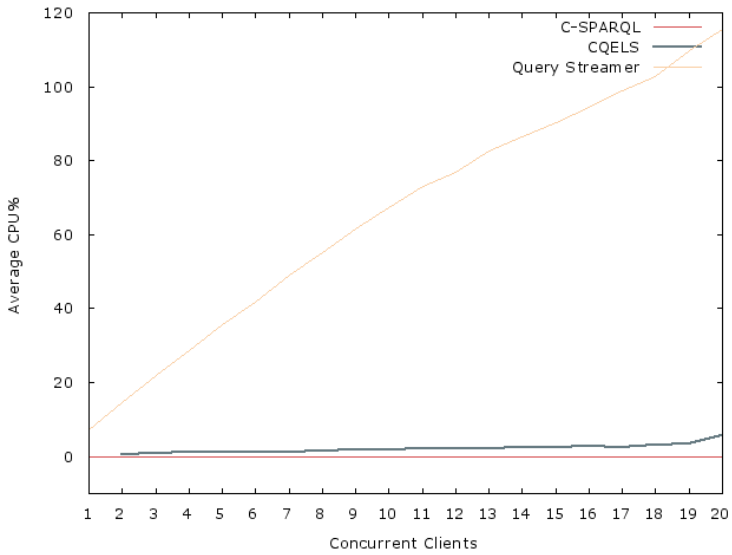


Figure 10.10: Combined client CPU usage for an increasing amount of clients for C-SPARQL, CQELS and the solution presented in this work. This shows that our solution does not perform very well at client side when different query streams are executed simultaneously. The CPU usage increases linearly as expected.

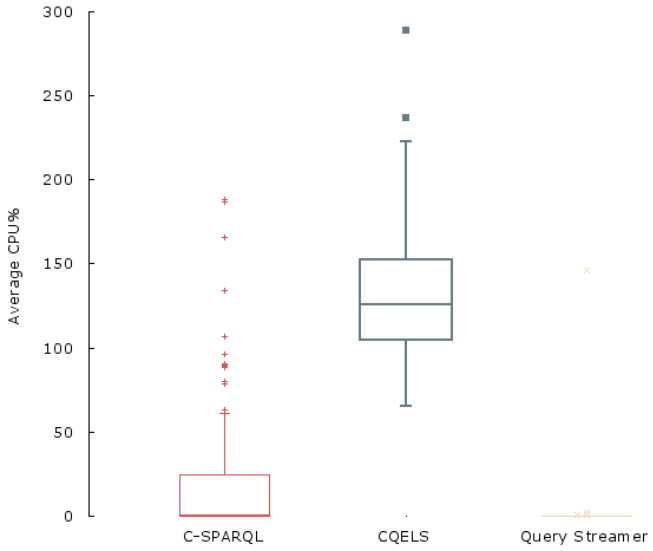


Figure 10.11: Average server CPU usage for 200 concurrent clients for C-SPARQL, CQELS and our solution. C-SPARQL and CQELS have a much higher overall CPU usage.

Chapter 11

Conclusion

In this thesis, we have researched and compared different approaches to continuously updating SPARQL queries for retrieving static and dynamic information, and staying up-to-date with this changing data. We investigated a solution for this using Triple Pattern Fragments and other Linked Data concepts to achieve more efficient, polling-based querying.

To pass on the volatility knowledge about facts to clients, we have researched different methods of time annotation for dynamic triples. These methods consist of reification, singleton properties, graphs and implicit graphs based on the properties of TPF. Each of these methods can be used with the annotation of either time intervals or expiration times.

The architecture that was developed is based on an extra layer on top of the TPF client. It can internally transform a regular query to two continuously executable queries that can be passed on to the regular TPF client. The first transformed query retrieves the time annotations of dynamic triple patterns so that the client knows when a new query needs to be executed to stay up-to-date with the data changes. The purpose of the second transformed query is to retrieve static information. A caching mechanism was built into this system to allow for this static information to be reused across different query executions. The TPF server needed no significant changes, except that its dynamic data must be time annotated.

An implementation of this architecture was created for all types of time annotations so that they could be compared with each other. This implementation was tested on performance for the different methods of time annotation in terms of processor usage and bandwidth. A naive approach to continuous querying was created using the regular TPF client and was compared with our implementation for performance. Finally, an experiment was created to compare the client and server performance between this implementation and two streaming-based SPARQL engines: C-SPARQL [7] and CQELS [10].

From our results, it was clear that graph-based annotation using expiration times is the most efficient way of handling dynamic data in the proposed framework, with implicit graphs using the properties of TPF being a very promising future research topic. This is because the implicit graphs approach allows clients who do not support time annotated facts to still retrieve all available information, expect for the time information itself. Our expectation that our approach would be more scalable in terms of server processor usage than other streaming-based SPARQL engines was confirmed. This is because TPF moves a large part of the query execution from the server to the client, which ensures that whoever wants to execute a query is also the one that will pay for the largest part of the execution cost. This is in essence a fundamental solution to increased availability.

When clients are very limited in terms of computational power, or when query endpoints are restriced to a low number of clients, then traditional approaches like C-SPARQL and CQELS are still more interesting than the solution presented in this work. This approach is aimed at use cases where a large number of clients concurrently have to query data without putting a too high load on the server.

Our results made it clear that polling-based continuously updating SPARQL querying is a valid alternative to the traditional approaches. While significantly reducing the required server computations per query, which leads to a higher endpoint availability. This work is only a first step for achieving a complete solution for this, since there are currently still some things that need to be tackled before this would be fully usable in practice. There are for example some parts of the query rewriter module which could be further improved to optimize the client querying efficiency. But it is also important that a generic method is devised for automatically annotating data with their volatility, because in this work we assumed that this data was already annotated with the required information. These will be further discussed in the next chapter.

Up until now, real-time data was either available with limited publicness for server performance reasons, or was published in formats which are not automatically machine-readable by the Semantic Web standards. Using the concept of Linked Data Fragments, we were able take the next step to publicly available real-time Linked Data.

Chapter 12

Future Work

This chapter will explain some aspects of this research which could be improved in future research. First, we will discuss some parts of the preprocessing step of our approach which can be improved. After that, the main query execution process will be handled. Finally some future opportunities for further evaluating this implementation are discussed.

12.1 Preprocessing

12.1.1 Multiple Dynamic Queries

In Chapter 8 we have explained that the **rewriter** module is responsible for splitting up the input query into a static and dynamic query. The dynamic query has time-annotated triple patterns which means that this query must be re-executed after a certain time. This time for re-execution is retrieved from either the expiration time or the upper bound of the time interval.

It was also explained that when multiple triple patterns with differing expiration times or time intervals are present in the dynamic query, the earliest of update times was taken to ensure that all results are up-to-date.

For the train-scheduling use case that was presented in Chapter 9, this approach works fine because these dynamic triple patterns require the same update frequency. But assume we have the query from Listing 12.1 which retrieves the time from a clock using three different triple patterns, each with a different update frequency. It is clear that each triple pattern will have a significantly different update frequency, for example the results for `?minute` will change 1 440 more frequently than those for `?day`. In the current solution, this complete dynamic query would be fully executed each minute, while it is known that the update frequency for two of the triple patterns are much lower.

```
PREFIX c: <http://example.org/clock#>

SELECT DISTINCT ?minute ?hour ?day
WHERE {
    c:clock c:minute ?minute .
    c:clock c:hour ?hour .
    c:clock c:day ?day .
}
```

Listing 12.1: SPARQL query for retrieving clock information with three dynamic triple patterns.

Instead of making a strict distinction between static and dynamic queries, it could be possible to have a set of queries which each have a different update frequency. The traditional static query would then become a query with an infinitely long update frequency. The dynamic query would be split up into different queries grouped by triple pattern update frequency. The caching mechanism could be generalized to not only be usable for fully static queries, but for queries with any update frequency.

This change would mean that the query from Listing 12.1 would result in three separate queries which each have a different update frequency. The results for `?hour` and `?day` would then be cached for their respective durations.

12.1.2 Query Splitting with `UNION` clauses

As was explained in Section 8.3.2, query splitting with multiple or nested `UNION` clauses can become impossible to perfectly split up to a disjunct static and dynamic query.

It might be possible to solve this issue using mathematical set theory which can be implemented using SPARQL algebra. This would require deeper interaction with the TPF client instead of just an extra layer on top of it. This is because our extra layer passes SPARQL queries to the TPF client and expects its results, but what we need is a more fine-grained way of managing which Triple Pattern Fragments are fetched and which are not. *Virtual queries* could be used to fetch mixed static and dynamic results but are still smart enough to cache the static data. These virtual queries could act as a proxy to the real triple pattern fragments retriever which is engrained into the TPF query executor to make sure that static results are retrieved from the cache if possible.

12.1.3 Implicit Graphs

Section 10.1.4 showed us that the implicit graph approach has a very inefficient rewriting phase. This is because of the issue that was already explained in Section 8.3.3, the fact that we have no easy way of knowing which triple patterns are dynamic without checking all matching triples.

What we need is a better way of determining whether or not triple patterns are static or

```
s p o .
_ p o .
s _ o .
_ _ o .
s p _ .
_ p _ .
s _ _ .
_ _ _ .
```

Listing 12.2: All possible triple patterns for the triple `s p o`.

dynamic, instead of just looking at the materialized triples and seeing if they have a time annotation. One possible approach for this is to add additional metadata to the TPF endpoint to indicate which triple patterns are dynamic. This however raises another issue, the question of which triple patterns should be added as metadata, because for each triple multiple corresponding triple patterns exist.

Listing 12.2 shows all 2^3 possibilities of triple patterns for a certain triple `s p o`. Future research might find a solution to determine which triple patterns should and should not be marked as dynamic. It is for example clear that the fully undetermined triple pattern `_ _ _` should only be marked as dynamic if all triples in the current dataset are dynamic. While the fully determined triple pattern `s p o` only depends on a single concrete triple to be marked as dynamic.

This extra metadata could significanly increase the dataset size, seeing as this dataset would approximately be eight times larger in the worst case. This number would of course be lower when some dynamic triples share a subject, predicate or object. It might also be possible to not explicitly store this metadata inside the store, but by dynamically resolving the triple pattern fragment URIs for these metadata and redirecting them to the correct metadata.

A second problem with implicit graphs is the extra query step during the streaming phase that was explained in Section 10.1.2 which explains the slightly higher execution times for this approach compared to regular graphs. Solving this issue would most likely require again a deeper interaction with the TPF client to avoid having multiple separate query executions which do not optimally make use of the query planner of the TPF client.

When these issues could be solved, implicit graphs would become a very interesting alternative to explicit graphs. Mainly because this wouldn't require graph support on the client or server, seeing as the graph structures become possible by using Triple Pattern Fragments.

12.2 Querying

12.2.1 Determining Expiration Times and Time Intervals

Throughout this work, we have always assumed that the data provider has sufficient knowledge to determine the expiration times and time intervals for dynamic facts. In Section 7.1.3, we discussed the use of mathematical equations to represent the volatility of data instead of the current methods. This would lead to the need of a generic approach for determining the change patterns of data, what could potentially be achieved by doing some kind of pattern matching on the partial history of the data changes.

If such a volatility prediction of data could somehow be achieved, this could also be used to determine each next expiration time and time interval for our dynamic facts. But this prediction is very use case specific, so a generic solution might not always be desired as it could introduce overhead when compared to more simple tailor-made approaches.

12.2.2 Concurrent Client-Query Optimization

Section 10.3 showed that our implementation causes a lineair increase in processor usage at the client when multiple different query streams are initiated. This is of course to be expected since the cost of executing queries has been largely moved from the server to the client.

In some cases, it might be possible to optimize these concurrent query executions at the client side. When for example these different queries are linear at the same dataset, parts of the execution could be reused. This would require all query executions to have a common context in which these reusable parts are stored, and what would again possibly require deeper interaction with the TPF client.

Local HTTP caches would be an easy first step to achieve this, as this would cause common HTTP requests to the endpoint to be only executed once.

12.2.3 TPF `FILTER` Support for Time Intervals

A fundamental idea of Linked Data Fragments is to limit the interfaces of endpoints to reduce the possibility of server overloading. TPF implements this by only allowing lookups on triple pattern fragments.

In this work we have presented an approach to annotate dynamic triples with time annotations. If we were using time intervals as annotation type, each fact could have multiple time versions in the dataset. For checking which version is currently valid, the client must retrieve each version of this fact, which leads to the problem of continuously slower query execution as discussed in Section 7.1.3 and experimentally confirmed in

Section 10.1.1.

We had already discussed the solution of ensuring an upper bound on how much fact versions can exist at the same time. An alternative to this limitation could be to expand the TPF endpoint interface to allow basic `FILTER` operations on the time annotations itself. If it were possible for the client to simply search for "*All departed trains between NOW - 10 minutes and NOW.*" as is possible with SPARQL$_{Stream}$ [9], a lot of bandwidth and processor time of the client could be saved. It would also have the nice side-effect of enabling historical searches, which opens up a whole new window of possibilities. This however, moves part of the complexity again to the server. But if the server was capable of somehow saving and looking up the results for these `FILTER`'s in an efficient manner, this could lead to an overall increase in efficiency.

12.2.4 Changing Background Data

In a similar work [68] about stream processing, researchers have developed a method for evaluating continuous queries over both static and dynamic data. They also take into account that static background data from other data sources can still change, and this change is reflected in local views on the server.

This is something that is not supported in the solution presented in our work. From the moment a client commences a query streaming, it will cache any static data it comes across. But this static data will remain the same for the whole remainder of this query streaming instance.

This related work [68] could be a good starting point for solving this issue. But it will not provide a complete solution, because we would need an additional mechanism to make sure the client is aware of any static data changes. One very simple solution could be to make all static data dynamic with a very low change frequency combined with allowing multiple dynamic queries as discussed in Section 12.1.1.

12.3 Evaluation

12.3.1 Influence of Different Queries

The evaluation that was done in Chapter 10 was limited to the query of a single use case, the one for retrieving information about train departures as could be seen in Listing 9.1. This query had two triple patterns retrieving dynamic data and three triples patterns for static data. In Section 10.1.3, we had already reasoned that when more triple patterns were to be used over static data, our caching mechanism would potentially lead to relatively lower bandwidth and executions times when compared to the naive approach.

Therefore, it would be interesting to do a similar evaluation with different query types.

Queries with much more dynamic data or triple patterns over dynamic data could be used, as well as queries with more static data or patterns over static data.

Currently, our dynamic data had a change frequency of ten seconds, experiments with varying frequencies might also produce interesting results about client and server performance.

12.3.2 Number of Concurrent Clients for Measuring Performance

For our experiments, we used the Virtual Wall [65] environment from iMinds which had a limited amount of machines available for testing. This is the reason why we only had physical 10 client machines plus 1 server in our performance tests. Even though Figure 10.8 already showed significant differences in server processor usage for our implementation compared to C-SPARQL [7] and CQELS [10], it showed that the processor usage for our implementation remained really low. It is not clear from this what the limitations of this approach could be.

One way to improve these results is to simply have more available machines to use as clients. Another way of improving this could be to increase the volatility of our dynamic data, but at the same time making sure that this frequency does not go below the minimum execution time of a single query stream execution for a certain time. Otherwise this would lead to overlapping query executions what would eventually lead to a client overload.

Bibliography

[1] Seth van Hooland and Ruben Verborgh. *Linked Data for Libraries, Archives and Museums*. Facet Publishing, June 2014. ISBN 978-1856049641.
URL `http://amzn.to/UUfjfN`

[2] Steve Harris, Andy Seaborne, and Eric Prud'hommeaux. SPARQL 1.1 Query Language. Recommendation, W3C, March 2013.
URL `http://www.w3.org/TR/2013/REC-sparql11-query-20130321/`

[3] Ruben Verborgh, Miel Vander Sande, Pieter Colpaert, Sam Coppens, Erik Mannens, and Rik Van de Walle. Web-Scale Querying through Linked Data Fragments. In *Proceedings of the 7th Workshop on Linked Data on the Web*. April 2014.
URL `http://events.linkeddata.org/ldow2014/papers/ldow2014_paper_04.pdf`

[4] Nuno Lopes, Antoine Zimmermann, Aidan Hogan, Gergely Lukácsy, Axel Polleres, Umberto Straccia, and Stefan Decker. RDF Needs Annotations. In *W3C Workshop on RDF Next Steps, Stanford, Palo Alto, CA, USA*. Citeseer, 2010.
URL `http://www.w3.org/2009/12/rdf-ws/papers/ws09`

[5] Gavin Carothers. RDF 1.1 N-Quads: A line-based syntax for RDF datasets. Recommendation, W3C, February 2014.
URL `http://www.w3.org/TR/2014/REC-n-quads-20140225/`

[6] Vinh Nguyen, Olivier Bodenreider, and Amit Sheth. Don't Like RDF Reification? Making Statements About Statements Using Singleton Property. In *Proceedings of the 23rd International Conference on World Wide Web*, WWW '14, pages 759–770. ACM, New York, NY, USA, 2014. ISBN 978-1-4503-2744-2. doi:10.1145/2566486.2567973.
URL `http://doi.acm.org/10.1145/2566486.2567973`

[7] Davide Francesco Barbieri, Daniele Braga, Stefano Ceri, and Michael Grossniklaus. An Execution Environment for C-SPARQL Queries. In *Proceedings of the 13th International Conference on Extending Database Technology*, EDBT '10, pages 441–452. ACM, New York, NY, USA, 2010. ISBN 978-1-60558-945-9. doi:10.1145/1739041.1739095.
URL `http://doi.acm.org/10.1145/1739041.1739095`

[8] Davide Barbieri, Daniele Braga, Stefano Ceri, Emanuele Della Valle, and Michael Grossniklaus. Stream Reasoning: Where We Got So Far. In *Proceedings of the*

NeFoRS2010 Workshop, co-located with ESWC2010. 2010.
URL http://wasp.cs.vu.nl/larkc/nefors10/paper/nefors10_paper_0.pdf

[9] Jean-Paul Calbimonte, Oscar Corcho, and Alasdair JG Gray. Enabling Ontology-Based Access to Streaming Data Sources. In *The Semantic Web–ISWC 2010*, pages 96–111. Springer, 2010.
URL http://link.springer.com/chapter/10.1007/978-3-642-17746-0_7

[10] Danh Le-Phuoc, Minh Dao-Tran, Josiane Xavier Parreira, and Manfred Hauswirth. A Native and Adaptive Approach for Unified Processing of Linked Streams and Linked Data. In *The Semantic Web–ISWC 2011*, pages 370–388. Springer, 2011.
URL http://iswc2011.semanticweb.org/fileadmin/iswc/Papers/Research_Paper/05/70310368.pdf

[11] Alejandro Rodrıguez, Robert McGrath, Yong Liu, James Myers, and IL Urbana-Champaign. Semantic Management of Streaming Data. *Proc. Semantic Sensor Networks*, 80, 2009.
URL http://ceur-ws.org/Vol-522/p12.pdf

[12] Tim Berners-Lee. Linked Data, July 2006.
URL http://www.w3.org/DesignIssues/LinkedData.html

[13] Tim Berners-Lee, James Hendler, Ora Lassila, et al. The Semantic Web. *Scientific american*, 284(5):28–37, 2001.
URL http://isel2918929391.googlecode.com/svn-history/r347/trunk/RPC/Slides/p01_theSemanticWeb.pdf

[14] Darko Anicic, Paul Fodor, Sebastian Rudolph, and Nenad Stojanovic. EP-SPARQL: A Unified Language for Event Processing and Stream Reasoning. In *Proceedings of the 20th International Conference on World Wide Web*, WWW '11, pages 635–644. ACM, New York, NY, USA, 2011. ISBN 978-1-4503-0632-4. doi: 10.1145/1963405.1963495.
URL http://doi.acm.org/10.1145/1963405.1963495

[15] Andre Bolles, Marco Grawunder, and Jonas Jacobi. *Streaming SPARQL-Extending SPARQL to Process Data Streams*. Springer, 2008.
URL http://link.springer.com/chapter/10.1007/978-3-540-68234-9_34

[16] Tim Berners-Lee. Linked Open Data. Talk, 2008.
URL http://www.w3.org/2008/Talks/0617-lod-tbl/#(3)

[17] JJ Carol and G Klyne. Resource Description Framework: Concepts and Abstract Syntax. Recommendation, W3C, February 2014.
URL http://www.w3.org/TR/2014/REC-rdf11-concepts-20140225/

[18] Richard Cyganiak, David Wood, and Markus Lanthaler. RDF 1.1 Concepts and Abstract Syntax: Replacing Blank Nodes with IRIs. Recommendation, W3C, February 2014.
URL http://www.w3.org/TR/2014/REC-rdf11-concepts-20140225/#section-skolemization

[19] Ruben Verborgh, Olaf Hartig, Ben De Meester, Gerald Haesendonck, Laurens De Vocht, Miel Vander Sande, Richard Cyganiak, Pieter Colpaert, Erik Mannens,

and Rik Van de Walle. Querying Datasets on the Web with High Availability. In *Proceedings of the 13th International Semantic Web Conference*. October 2014. URL http://linkeddatafragments.org/publications/iswc2014.pdf

[20] Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez. Semantics and Complexity of SPARQL. *ACM Trans. Database Syst.*, 34(3):16:1–16:45, September 2009. ISSN 0362-5915. doi:10.1145/1567274.1567278. URL http://doi.acm.org/10.1145/1567274.1567278

[21] Tim Berners-Lee, Christian Bizer, and Tom Heath. Linked data-the story so far, 2009. URL http://www.dagstuhl.de/Materials/Files/09/09271/09271.BizerChristian.Slides.pdf

[22] Tim Bray, Jean Paoli, C Michael Sperberg-McQueen, Eve Maler, and François Yergeau. EXtensible Markup Language (XML). *World Wide Web Consortium Recommendation REC-xml-19980210*, 1998. URL http://www.w3.org/TR/1998/REC-xml-19980210

[23] Mike Amundsen. *Hypermedia APIs with HTML5 & Node*. O'Reilly Media, November 2011. ISBN 978-1-4493-0657-1.

[24] David Beckett, Tim Berners-Lee, Eric Prudhommeaux, and Gavin Carothers. Turtle-terse RDF triple language. Recommendation, W3C, February 2014. URL http://www.w3.org/TR/2014/REC-turtle-20140225/

[25] Frank Manola, Eric Miller, and Brian McBride. RDF 1.1 Primer. Working group note, W3C, June 2014. URL http://www.w3.org/TR/2014/NOTE-rdf11-primer-20140624/#section-turtle-family

[26] Gavin Carothers and Andy Seaborne. RDF 1.1 TriG. Recommendation, W3C, February 2014. URL http://www.w3.org/TR/2014/REC-trig-20140225/

[27] Dan Brickley, Ramanathan V Guha, and Brian McBride. RDF vocabulary description language 1.0: RDF Schema. W3C Recommendation (2004). 2004. URL http://www.w3.org/tr/2004/rec-rdf-schema-20040210

[28] Deborah L McGuinness, Frank Van Harmelen, et al. OWL Web Ontology Language Overview. *W3C recommendation*, 10(10):2004, 2004. URL http://202.119.112.136/pweb/~zhuoming/teachings/MOD/N4/Readings/5.3-B1.pdf

[29] Peter Lambert. Internettechnology - Semantic Web Technology, 2012-2013.

[30] C. Gutierrez, C.A Hurtado, and A Vaisman. Introducing Time into RDF. *Knowledge and Data Engineering, IEEE Transactions on*, 19(2):207–218, Feb 2007. ISSN 1041-4347. doi:10.1109/TKDE.2007.34. URL http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4039284

[31] Claudio Gutierrez, Carlos Hurtado, and Alejandro Vaisman. Temporal RDF. In *The Semantic Web: Research and Applications*, pages 93–107. Springer, 2005. URL http://link.springer.com/chapter/10.1007/11431053_7

[32] Eric PrudHommeaux, Andy Seaborne, et al. SPARQL Query Language for RDF. *W3C recommendation*, 15, 2008.
URL http://www.w3.org/TR/rdf-sparql-query/

[33] Roy Fielding, Jim Gettys, Jeffrey Mogul, Henrik Frystyk, Larry Masinter, Paul Leach, and Tim Berners-Lee. RFC 2616, Hypertext Transfer Protocol–http/1.1, 1999. *Internet Engineering Task Force*, 2009.
URL http://www.rfc.net/rfc2616.html

[34] Orri Erling and Ivan Mikhailov. RDF Support in the Virtuoso DBMS. In *Networked Knowledge-Networked Media*, pages 7–24. Springer, 2009.
URL http://link.springer.com/chapter/10.1007/978-3-642-02184-8_2

[35] Michael Grobe. RDF, Jena, SPARQL and the 'Semantic Web'. In *Proceedings of the 37th annual ACM SIGUCCS fall conference*, pages 131–138. ACM, 2009.
URL http://dl.acm.org/citation.cfm?id=1629525

[36] Carlos Buil-Aranda, Aidan Hogan, Jürgen Umbrich, and Pierre-Yves Vandenbussche. SPARQL Web-Querying Infrastructure: Ready for Action? In *The Semantic Web–ISWC 2013*, pages 277–293. Springer, 2013.
URL http://link.springer.com/chapter/10.1007/978-3-642-41338-4_18

[37] Manish Marwah, Paulo Maciel, Amip Shah, Ratnesh Sharma, Tom Christian, Virgilio Almeida, Carlos Araújo, Erica Souza, Gustavo Callou, Bruno Silva, et al. Quantifying the Sustainability Impact of Data Center Availability. *ACM SIGMETRICS Performance Evaluation Review*, 37(4):64–68, 2010.
URL http://dl.acm.org/citation.cfm?id=1773405

[38] Ruben Verborgh. Linked Data Fragments. Unofficial draft, Hydra W3C Community Group, August 2014.
URL http://www.hydra-cg.com/spec/latest/linked-data-fragments/

[39] Ruben Verborgh. Triple Pattern Fragments. Unofficial draft, Hydra W3C Community Group, August 2014.
URL http://www.hydra-cg.com/spec/latest/triple-pattern-fragments/

[40] M. Nottingham. Feed Paging and Archiving. RFC 5005, Internet Engineering Task Force, September 2007.
URL http://www.rfc-editor.org/rfc/rfc5005.txt

[41] Markus Lanthaler and Christian Gütl. Hydra: A Vocabulary for Hypermedia-Driven Web APIs. In *LDOW*. Citeseer, 2013.
URL http://www.markus-lanthaler.com/research/
hydra-a-vocabulary-for-hypermedia-driven-web-apis.pdf

[42] Keith Alexander and Michael Hausenblas. Describing Linked Datasets - on the Design and Usage of VOID, the Vocabulary of Interlinked Datasets. In *In Linked Data on the Web Workshop (LDOW 09), in conjunction with 18th International World Wide Web Conference (WWW 09*. 2009.
URL http://www.w3.org/TR/void/

[43] Open Knowledge Foundation Belgium. iRail API.
URL https://irail.be/

[44] Mokrane Bouzeghoub. A Framework for Analysis of Data Freshness. In *Proceedings of the 2004 International Workshop on Information Quality in Information Systems*, IQIS '04, pages 59–67. ACM, New York, NY, USA, 2004. ISBN 1-58113-902-0. doi:10.1145/1012453.1012464.
URL http://doi.acm.org/10.1145/1012453.1012464

[45] Matthias Jarke, Manfred A Jeusfeld, Christoph Quix, and Panos Vassiliadis. Architecture and Quality in Data Warehouses: An Extended Repository Approach. *Information Systems*, 24(3):229–253, 1999.
URL http://www.sciencedirect.com/science/article/pii/S0306437999000174

[46] Donald Ballou, Richard Wang, Harold Pazer, and Giri Kumar.Tayi. Modeling Information Manufacturing Systems to Determine Information Product Quality. *Management Science*, 44(4):pp. 462–484, 1998. ISSN 00251909.
URL http://www.jstor.org/stable/2634609

[47] A Segev and W. Fang. Currency-Based Updates to Distributed Materialized Views. In *Data Engineering, 1990. Proceedings. Sixth International Conference on*, pages 512–520. Feb 1990. doi:10.1109/ICDE.1990.113505.
URL http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=113505

[48] Richard Y Wang and Diane M Strong. Beyond accuracy: What Data Quality Means to Data Consumers. *Journal of management information systems*, pages 5–33, 1996.
URL http://www.jstor.org/stable/40398176

[49] Joachim Hammer, Hector Garcia-Molina, Jennifer Widom, Wilburt Labio, and Yue Zhuge. The Stanford Data Warehousing Project. 1995.
URL http://ilpubs.stanford.edu:8090/76/

[50] Manu Sporny, Dave Longley, Gregg Kellogg, Markus Lanthaler, and Niklas Lindstrm. JSON-LD 1.0: A JSON-based Serialization for Linked Data. Recommendation, W3C, January 2014.
URL http://www.w3.org/TR/2014/REC-json-ld-20140116/

[51] Jeremy J. Carroll, Christian Bizer, Pat Hayes, and Patrick Stickler. Named Graphs, Provenance and Trust. In *Proceedings of the 14th International Conference on World Wide Web*, WWW '05, pages 613–622. ACM, New York, NY, USA, 2005. ISBN 1-59593-046-9. doi:10.1145/1060745.1060835.
URL http://doi.acm.org/10.1145/1060745.1060835

[52] E. Della Valle, S. Ceri, F. van Harmelen, and D. Fensel. It's a Streaming World! Reasoning upon Rapidly Changing Information. *Intelligent Systems, IEEE*, 24(6):83–89, Nov 2009. ISSN 1541-1672. doi:10.1109/MIS.2009.125.
URL http://www.few.vu.nl/~frankh/postscript/IEEE-IS09.pdf

[53] Arvind Arasu, Brian Babcock, Shivnath Babu, John Cieslewicz, Mayur Datar, Keith Ito, Rajeev Motwani, Utkarsh Srivastava, and Jennifer Widom. STREAM: The Stanford Data Stream Management System. *Book chapter*, 2004.
URL http://ilpubs.stanford.edu:8090/641/1/2004-20.pdf

[54] Shivnath Babu and Jennifer Widom. Continuous Queries over Data Streams. *ACM Sigmod Record*, 30(3):109–120, 2001.
URL http://dl.acm.org/citation.cfm?id=603884

[55] Davide Francesco Barbieri, Daniele Braga, Stefano Ceri, Emanuele Della Valle, and Michael Grossniklaus. Querying RDF Streams with C-SPARQL. *SIGMOD Rec.*, 39(1):20–26, September 2010. ISSN 0163-5808. doi:10.1145/1860702.1860705.
URL http://doi.acm.org/10.1145/1860702.1860705

[56] Darko Anicic, Paul Fodor, Sebastian Rudolph, Roland Sthmer, Nenad Stojanovic, and Rudi Studer. A Rule-Based Language for Complex Event Processing and Reasoning. In Pascal Hitzler and Thomas Lukasiewicz, editors, *Web Reasoning and Rule Systems*, volume 6333 of *Lecture Notes in Computer Science*, pages 42–57. Springer Berlin Heidelberg, 2010. ISBN 978-3-642-15917-6. doi: 10.1007/978-3-642-15918-3_5.
URL http://dx.doi.org/10.1007/978-3-642-15918-3_5

[57] Jess Barrasa, scar Corcho, and Asuncin Gmez-prez. R2O, an Extensible and Semantically based Database-to-Ontology Mapping Language. In *in In Proceedings of the 2nd Workshop on Semantic Web and Databases(SWDB2004*, pages 1069–1070. Springer, 2004.
URL http://www.cs.man.ac.uk/~ocorcho/documents/SWDB2004_BarrasaEtAl.pdf

[58] ChristianY.A. Brenninkmeijer, Ixent Galpin, AlvaroA.A. Fernandes, and NormanW. Paton. A Semantics for a Query Language over Sensors, Streams and Relations. In Alex Gray, Keith Jeffery, and Jianhua Shao, editors, *Sharing Data, Information and Knowledge*, volume 5071 of *Lecture Notes in Computer Science*, pages 87–99. Springer Berlin Heidelberg, 2008. ISBN 978-3-540-70503-1. doi: 10.1007/978-3-540-70504-8_9.
URL http://dx.doi.org/10.1007/978-3-540-70504-8_9

[59] Daniel J. Abadi, Adam Marcus, Samuel R. Madden, and Kate Hollenbach. Scalable Semantic Web Data Management Using Vertical Partitioning. In *Proceedings of the 33rd International Conference on Very Large Data Bases*, VLDB '07, pages 411–422. VLDB Endowment, 2007. ISBN 978-1-59593-649-3.
URL http://dl.acm.org/citation.cfm?id=1325851.1325900

[60] Norbert Lanzanasto, Srdjan Komazec, and Ioan Toma. Reasoning over Real Time Data Streams. *ENVISION Deliverable D*, 4.
URL http://www.envision-project.eu/wp-content/uploads/2012/11/D4.8-1.0.pdf

[61] Ying Zhang, PhamMinh Duc, Oscar Corcho, and Jean-Paul Calbimonte. SRBench: A Streaming RDF/SPARQL Benchmark. In Philippe Cudr-Mauroux, Jeff Heflin, Evren Sirin, Tania Tudorache, Jrme Euzenat, Manfred Hauswirth, JosianeXavier Parreira, Jim Hendler, Guus Schreiber, Abraham Bernstein, and Eva Blomqvist, editors, *The Semantic Web ISWC 2012*, volume 7649 of *Lecture Notes in Computer Science*, pages 641–657. Springer Berlin Heidelberg, 2012. ISBN 978-3-642-35175-4. doi:10.1007/978-3-642-35176-1_40.
URL http://dx.doi.org/10.1007/978-3-642-35176-1_40

[62] David Carlisle, Patrick Ion, and Robert Miner. Mathematical Markup Language (MathML). Recommendation, W3C, April 2014.
URL http://www.w3.org/TR/2014/REC-MathML3-20140410/

[63] Ken Wenzel and Heiner Reinhardt. Mathematical Computations for Linked Data Applications with OpenMath. In *Joint Proceedings of the 24th Workshop on OpenMath and the 7th Workshop on Mathematical User Interfaces (MathUI)*, page 38. Citeseer, 2012.
URL http://ceur-ws.org/Vol-921/openmath-01.pdf

[64] Joycent, Inc. Node.js.
URL http://nodejs.org/

[65] iMinds iLab.t. Virtual Wall: wired networks and applications.
URL http://ilabt.iminds.be/virtualwall

[66] Chen Levan. CQELS engine: Instructions on experimenting CQELS.
URL https://code.google.com/p/cqels/wiki/CQELS_engine

[67] StreamReasoning. Continuous SPARQL (C-SPARQL) Ready To Go Pack.
URL http://streamreasoning.org/download

[68] Soheila Dehghanzadeh, Daniele Dell'Aglio, Shen Gao, Emanuele Della Valle, Alessandra Mileo, and Abraham Bernstein. Approximate Continuous Query Answering Over Streams and Dynamic Linked Data Sets. In *15th International Conference on Web Engineering*. s.n., Switzerland, June 2015.
URL http://dx.doi.org/10.5167/uzh-110296