Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science 137 (2018) 43–54

Procedia Computer Science

www.elsevier.com/locate/procedia

SEMANTiCS 2018 – 14th International Conference on Semantic Systems

On the Semantics of TPF-QS towards Publishing and Querying RDF Streams at Web-scale

Ruben Taelman[a], Riccardo Tommasini[b], Joachim Van Herwegen[a], Miel Vander Sande[a], Emanuele Della Valle[b], Ruben Verborgh[a]

[a]*Ghent University – imec – IDLab, Belgium*

[b]*Dipartimento di Elettronica, Informazione e Bioingegneria, Politecnico of Milano, Italy*

Abstract

RDF Stream Processing (RSP) is a rapidly evolving area of research that focuses on extensions of the Semantic Web in order to model and process Web data streams. While state-of-the-art approaches concentrate on server-side processing of RDF streams, we investigate the Triple Pattern Fragments Query Streamer (TPF-QS) method for server-side publishing of RDF streams, which moves the workload of continuous querying to clients. We formalize TPF-QS in terms of the RSP-QL reference model in order to formally compare it with existing RSP query languages. We experimentally validate that, compared to the state of the art, the server load of TPF-QS scales better with increasing numbers of concurrent clients in case of simple queries, at the cost of increased bandwidth consumption. This shows that TPF-QS is an important first step towards a viable solution for Web-scale publication and continuous processing of RDF streams.

Keywords: Linked Data; RDF stream processing; continuous querying; TPF-QS; RSP-QL; SPARQL

1. Introduction

The problem of processing streams received a lot of attention by the Semantic Web community in the last 10 years [12]. Under the label of Stream Reasoning [10], several approaches were proposed, spanning from query answering to temporal and incremental reasoning. Among these solutions, RDF Stream Processing (RSP) engines [6, 18, 9, 2] process unbounded sequences of non-decreasing timestamped RDF data on-the-fly.

RSP engines are designed to handle large volumes of heterogeneous data in a short time, whereas SPARQL engines [17] allow querying over datasets which change never or very slowly. There exists however a range of use-cases inbetween these two extremes, where neither highly volatile streams or nearly static datasets apply.

An example of such a use-case can be found in the CityBench RSP benchmark [1]. In this case, the average number of cars that enter and exit a parking area in a given time interval is monitored. Data are produced by a sensor network deployed over the parking area where new observations are streamed out every 5 minutes.

E-mail address: ruben.taelman@ugent.be

10.1016/j.procs.2018.09.005

With the current state of the art, we have to either keep the server always active with an RSP engine for this slow stream, or we have to use a traditional SPARQL solution that lacks the temporal dimension. Clearly, these solutions are both sub-optimal and they suffer from scalability problems for a large number of clients that need to query the stream. This is because RSP engines have to maintain open connections with each client to push new results, whereas the high complexity of SPARQL queries can lead to low availability in SPARQL endpoints [8].

In previous work, which we further elaborate on in the next section and Section 3, we introduced Triple Pattern Fragments Query Streamer (TPF-QS) [23] to provide a different trade-off between server and client load for Web-scale querying of RDF streams. Instead of evaluating continuous queries server-side and pushing results to clients, which is the case with most of the RSP engines today, TPF-QS evaluates these queries client-side, using a low-cost data interface on the server.

Now, we support this contribution by formalizing the operational semantics of TPF-QS using RSP-QL [11] (Section 4). This enables us to formally compare its evaluation with other RSP engines (Section 5). After that, we extensively evaluate TPF-QS using CityBench [1] (Section 6), an RSP benchmark with real-world data. Finally, we discuss these results and its conclusions in Section 7.

2. Related Work

In this section, we first present the related work on querying and publishing Linked Data, which is required for formalizing our approach. After which we present approaches for querying and publishing within the context of RDF streams that we will compare our approach to in this work.

2.1. Linked Data Querying and Publishing

RDF is a data model used to represent Linked Data on the Web. The SPARQL protocol [14] enables query execution on this data using the SPARQL syntax [17]. $\mathcal{U}$, $\mathcal{B}$, $\mathcal{L}$, and $\mathcal{V}$ respectively refer to the sets of all URIs, blank nodes, literals, and variables. The basis of the RDF model is the *RDF statement* or *RDF triple* τ , which is defined as $\tau \in (\mathcal{U} \cup \mathcal{B}) \times \mathcal{U} \times (\mathcal{U} \cup \mathcal{B} \cup \mathcal{L})$. RDF triples are the basis of the RDF model; a set of RDF statements is called an *RDF graph*.

A *triple pattern* ρ enables pattern matching over triples, and is defined as $\rho \in (\mathcal{U} \cup \mathcal{V}) \times (\mathcal{U} \cup \mathcal{V}) \times (\mathcal{U} \cup \mathcal{L} \cup \mathcal{V})$. A set of triple patterns is called a *Basic Graph Pattern* (BGP), which is the basis of a SPARQL query. A SPARQL query is always targeted at a certain *RDF dataset*, which is defined by one *default graph* and zero or more *named graphs*. The default graph is the default active graph for pattern matching, which can be changed to any of the named graphs. The result of a SPARQL query is called a *solution mapping*, which is a partial function $\mu : \mathcal{V} \mapsto \mathcal{U} \cup \mathcal{B} \cup \mathcal{L}$.

Research has shown that the average public SPARQL endpoint is down for more than 1.5 days each month [8]. The *Triple Pattern Fragments* (TPF) interface [24] was introduced as a possible solution to this availability problem, moving part of the query evaluation complexity from the server to the client. The server exposes a low-cost interface that only accepts triple pattern queries, which are simpler for the server to evaluate than full SPARQL queries. Since caching of such coarse fragments is more effective, server load can be further reduced when multiple clients execute queries. A TPF client is a client-side SPARQL engine that is able to evaluate SPARQL queries at the client by consuming data from one or more TPF interfaces. Results show that this technique significantly reduces server load, at the cost of increased query execution times and bandwidth usage [24]. The TPF-QS approach has been built on top of the TPF approach, with the aim of reducing server load in the context of RDF streams.

2.2. RDF stream Querying and Publishing

RDF Stream Processing (RSP) engines are used to query RDF streams, which are unbounded sequences of data encoded in RDF. Data Stream Management Systems (DSMS) [4] and Complex Event Processing (CEP) [19] are stream processing paradigms that can categorize RSP engines. The C-SPARQL [6] engine (which we will refer to as C-SPARQL), CQELS [18] and Morph$_{stream}$ (implementing the SPARQL$_{stream}$ language [9]) are examples of DSMS systems that focus on the continuous evaluation of queries over RDF streams. Continuous querying in those systems is possible by the means of SPARQL 1.1 extensions for continuous semantics [3]. Differences in the window operator implementation, and query evaluation strategies, distinguish the operational semantics of CQELS and C-SPARQL [11].

ETALIS [2] is an RSP engine that implements the EP-SPARQL language and combines RDF streams processing with CEP operators. In contrast to CQELS or C-SPARQL, it has no concept of windowing, but RDF statements are annotated with

```
_:g0 { ex:sensor1 sao:hasValue "20.1" }
_:g0 tmp:initial "2017-11-01T10:00:00"^^xsd:dateTime;
     tmp:final   "2017-11-01T10:01:00"^^xsd:dateTime.
_:g1 { ex:sensor1 sao:hasValue "20.2" }
_:g1 tmp:initial "2017-11-01T10:01:00"^^xsd:dateTime;
     tmp:final   "2017-11-01T10:02:00"^^xsd:dateTime.
```

Listing 1: Time interval-based representation of a stream that is produced by thermometer `ex:sensor1` at a frequency of one minute.

two timestamps that indicate their interval of validity. By means of an extended version of SPARQL 1.0, patterns can be detected over RDF streams and more complex events can be constructed.

CQELS, Morph$_{stream}$, and ETALIS do not address the problem of publishing RDF streams, but instead delegate it to the user. Barbieri et al. [5] use C-SPARQL to publish streams as RDF streams by the means of a RESTful API that controls the window. TripleWave [20] is a framework for publishing RDF streams on Web. It tackles common use cases for RDF stream publishing: *Conversion* uses the RML mappings language [13], to annotate existing raw streams into RDF streams; *Replay* streams historical streams that are stored as RDF datasets. Both approaches [5, 20] describe a stream using two types of named graphs referenced using URIs according to the Linked Data Principles: *(iGraphs)* Instantaneous Graphs represent a content unit in the RDF stream. *(sGraph)* the Stream Graph contains metadata about iGraphs.

3. Background

In this section, we present those background concepts required for the rest of the work. For brevity, we omit or simplify some definitions. We invite the reader to refer to the original papers for more precise and complete references.

3.1. Client-side RDF Stream Processing with TPF-QS

In previous work, we introduced TPF-QS [23], which is based on the TPF framework. Within TPF-QS, RDF streams are exposed, together with static background knowledge, through the TPF interface, which enables continuous client-side processing. TPF-QS aims to close the gap between the processing of high-velocity streams and static datasets by considering slowly evolving streams with a predictable rate or historical streams. Assuming storage is cheap, these RDF streams are stored instead of streamed, so that clients can retrieve the data in a pull-based manner, which reduces publication costs since the server is not responsible for pushing data to all registered clients. All RDF statements within the stream are annotated with a time interval, which indicates for how long these statements remain valid, which must be known at insertion time. An example of such a stream can be found in Listing 1.

The client-side TPF-QS engine[1] accepts query registrations using regular SPARQL 1.1 syntax [17], continuously evaluates them, and streams back the results. As a first step, the engine splits the incoming query into two separate queries. Based on the stream's metadata, one is targeted at an RDF stream and one targeted at the static background knowledge. Thereby, it can cache parts of the background knowledge that are assumed not to change over the course of the query evaluation. For example, assuming the stream from Listing 1, the query Listing 2 is split into the queries from Listings 3 and 4.

Next, TPF-QS starts by evaluating the static background knowledge query, which in our example is results in the binding `ex:sensor1` for `?sensor`. After that, the query targeted against the stream is materialized using these bindings. For example, the query from Listing 4 would be materialized to the query from Listing 5. TPF-QS now evaluates this materialized query, and retrieves the result that has a time interval that applies to the current time. The expiration times of these time intervals determine the time at which the materialized query should be re-evaluated. The stream from Listing 1 will therefore lead to an evaluation frequency of one minute.

In previous work, we have demonstrated that this technique can be used as part of a low-cost continuous sensor observation publication pipeline [21], and previous results show that this approach is able to achieve a lower server load compared to server-side RSP engines (C-SPARQL and CQELS) for an increasing number of concurrent clients for simple queries over a small dataset [23].

[1] The open source engine is available under a public license: https://github.com/LinkedDataFragments/QueryStreamer.js

```
SELECT ?temperature WHERE {
  ?sensor a ex:Thermometer.
  ?sensor sao:hasValue ?temperature.
}
```

Listing 2: Query for retrieving thermometer measurements

```
SELECT ?sensor WHERE {
  ?sensor a ex:Thermometer.
}
```

Listing 3: Query for retrieving thermometers from the static background knowledge

```
SELECT ?temperature ?initial ?final ?sensor WHERE {
  _:g {
      ?sensor sao:hasValue ?temperature
  }
  _:g tmp:initial ?initial;
      tmp:final   ?final.
}
```

Listing 4: Query for retrieving time-annotated sensor measurements from the stream.

```
SELECT ?temperature ?initial ?final WHERE {
  _:g {
    ex:sensor1 sao:hasValue ?temperature
  }
  _:g tmp:initial ?initial;
      tmp:final   ?final.
}
```

Listing 5: Materialized query for retrieving time-annotated sensor measurements.

3.2. The RSP-QL RDF Stream Processing Query Model

As listed in Section 2.2, several RSP engines were developed independently in the past [18, 6, 2, 9], resulting in a high variety in stream handling and query evaluation. RSP-QL [11] is a unifying model thats captures the differences between DSMS systems [4] such as CQELS, C-SPARQL and SPARQL$_{stream}$.

RSP-QL extends RDF and SPARQL, adding the time semantics as initially proposed in CQL [3]. A time unit is the constant difference between two consecutive time instants $(t_{i+1} - t_i)$, which belong to an infinite, discrete, and ordered sequence T.

An RDF stream $S_{\text{RSP-QL}}$ is an unbounded sequence of pairs (τ, t) where τ is an RDF statement and $t \in T$ is a time instant. t denotes a partial ordering because it is non-decreasing.

Querying RDF streams requires extending the concept of an RDF dataset against which a query is evaluated. An RSP-QL dataset comprises an optional default graph, zero or more named graphs, and zero or more (named) *time-varying graphs*, i.e., a function that maps time instants $t \in T$ to *instantaneous* RDF graphs.

The infinite nature of streams calls for specialized operators like time-based windows, that use opening (o) and closing (c) time instants to select subset of a stream $S_{\text{RSP-QL}}$, with a window defined by the interval $(o, c]$. A time-based *sliding* window is defined as $\mathbb{W}(\alpha, \beta, t^0)$, where α is the width of the window in time units, β is the slide parameter that defines how often the selection happens, and t^0 is the time instant at which $\mathbb{W}$ starts to operate. The present window W_p is defined as the window that is present in relation to the current moment in time. A time-based sliding window $\mathbb{W}$ takes as an input a stream $S_{\text{RSP-QL}}$ and produces a time-varying graph $G_{\mathbb{W}}$.

The query evaluation repeats continuously upon a potentially infinite and stream-dependent sequence of time instants ET. These are determined by a policy P that can be a combination of one or more boolean conditions, called *strategies*. *Content Change (CC)*: the window reports if the content changes; *Window Close (WC)*: the window reports if the active window closes, i.e., when the current timestamp falls outside of the active window; *Non-empty Content (NC)*: the window reports if the active window is not empty, i.e., if the active window contains at least one statement; *Periodic (P)*: the window reports at regular intervals.

Finally, in order to report a stream of consecutive RSP-QL query results in a certain format, three streaming operators exist. They introduce a temporal dimension in the data by timestamping the solution mappings. *RStream* annotates an input sequence of solution mappings with the evaluation time; *IStream* streams the difference between the answer of the current evaluation and the one of the previous iteration; *DStream* streams the part of the answer at the previous iteration that is not in the current one.

4. Semantics of TPF-QS Querying

The client-side TPF-QS engine is an extension of the TPF client. Whereas TPF clients can only query static datasets, TPF-QS clients are able to use TPF interfaces to consume the RDF streams and background knowledge. In this section, we formalize its operational semantics using RSP-QL. We start by formalizing the stream publishing mechanism, followed by the client-side windowing operation. Finally, we discuss an operational example.

4.1. Stream Publication

Client-side TPF-QS engines can consume RDF streams exposed through a TPF interface to (continuously) evaluate queries. Similar to a RSP-QL stream, we represent a TPF-QS stream $S_{\text{TPF-QS}}$ as an unbounded sequence of pairs $(\tau, [t_1, t_2])$; where τ is an RDF statement and $[t_1, t_2]$ a time interval [16], with time points $t_1, t_2 \in T$ and $t_1 \leq t_2$, which denotes the closed interval from t_1 to t_2. Several triple-level time-annotation strategies exist to serialize this TPF-QS stream to RDF, as described in previous work [23].

As a special case of time intervals, expiration times are defined as the last time point at which a statement is valid [23]. An RDF statement τ with expiration time e is equivalent to that same statement τ with as time interval $[0, e]$. These expiration times can be used when only the last version of certain observations needs to be stored.

In order to conform with the definition of an RDF stream within the RSP-QL model, we can implicitly transform any time interval $[a, b]$ to a consecutive sequence of time instants $a \ldots b$ [7]. We therefore introduce the TpfqsToRspql function to transform a TPF-QS stream to a RSP-QL stream.

Definition 1. $\text{TpfqsToRspql}(S_{\textit{TPF-QS}}) = \{ (\tau, t) \mid \exists t_1, t_2 : (\tau, [t_1, t_2]) \in S_{\textit{TPF-QS}} \wedge t \in [t_1, t_2] \}$, *with* $S_{\textit{TPF-QS}}$ *a TPF-QS stream. Additionally, the resulting set is ordered by* $\prec$ *with* $(\tau_1, t_1) \prec (\tau_2, t_2)$ *if* $t_1 < t_2$.

4.2. Client-side Windowing

Within the TPF-QS approach, streams are stored server-side, while windowing and query evaluation happens client-side. Therefore, we introduce the following definitions.

Window Start TPF-QS allows users to optionally pick a starting time instant t^0 different from the current time. This aspect is unique among state-of-the-art RSP engines and enables TPF-QS to query historical streams.

Tumbling Time-based Window The TPF-QS engine applies a time-based tumbling window with a width α of one time unit, such that for an opening time o_p and closing time c_p we always have $c_p - o_p = 1$. A window opening at o_p will always be defined by the interval $(o_p, c_p]$. In this case, the resulting time-varying graph $G_{\mathbb{W}}$ will therefore contain exactly one instantaneous RDF graph G_W, which simplifies query evaluation using a non-continuous SPARQL engine, as processing can happen atemporally.

Windowing Strategy TPF-QS has two configurable windowing strategies that characterize its operational semantics: *Periodic* or *Mapping Expire* (ME). The latter is the default policy for the TPF-QS engine.

As explained in Section 3, when working with the *Periodic* strategy a window reports regularly over time. In this case, TPF-QS requires its user to specify a time period p that is used as the time difference between two evaluation times.

The *Mapping Expire* strategy is specific to TPF-QS: the window reports when the validity of an RDF statement that was used in the last solution mapping expires. This new strategy is possible because of the additional validity duration metadata on RDF statements that are represented by time intervals in TPF-QS streams. In the following, we provide the formal definition of this strategy w.r.t the RSP-QL model, i.e. how it characterizes the sequence of evaluation time instants ET. To this extent, we first formalize the notions of *window on a TPF-QS stream*; *valid time intervals* of a statement; *active time interval* for a statement and, finally, *window expiration*.

Definition 2. *The window* W_t *on stream* $S_{\textit{TPF-QS}}$ *is the set of triples valid at time* t: $W_t = \{ \tau \mid \exists t_1, t_2 : (\tau, [t_1, t_2]) \in S_{\textit{TPF-QS}} \wedge t \in [t_1, t_2] \}$

Definition 3. *The set of all time intervals of an RDF statement* τ *in a TPF-QS stream* $S_{\textit{TPF-QS}}$ *is* $\text{validTimeIntervals}(\tau, S_{\textit{TPF-QS}}) = \{ [t_1, t_2] \mid (\tau, [t_1, t_2]) \in S_{\textit{TPF-QS}} \}$.

Definition 4. *The set of all time intervals of an RDF statement* τ *in a TPF-QS stream* $S_{\textit{TPF-QS}}$ *valid at time* t *is* $\text{activeTimeIntervals}(\tau, S_{\textit{TPF-QS}}, t) = \{ [t_1, t_2] \mid t \in [t_1, t_2] \wedge [t_1, t_2] \in \text{validTimeIntervals}(\tau, S_{\textit{TPF-QS}}) \}$.

We can now define the *expiration* of a statement in a stream given a time instant.

Definition 5. *For τ an RDF statement, $S_{TPF\text{-}QS}$ a TPF-QS stream and t a timestamp, let $A = \text{activeTimeIntervals}(\tau, S_{TPF\text{-}QS}, t)$. We then define*

$$\text{nextExpiration}(\tau, S_{TPF\text{-}QS}, t) = \begin{cases} \max\{\, t_2 \mid [t_1, t_2] \in A, \textit{with the smallest possible } t_1 \,\} & \textit{if } A \neq \emptyset \\ \bot & \textit{else} \end{cases}$$

This means we take the interval with the largest t_2 and smallest t_1 if there are multiple. Using these definitions we can now define the expiration time of a Basic Graph Pattern as the earliest one of the relevant triples in a window expires.

Definition 6. *The expiration of a Basic Graph Pattern BGP with respect to a window W_t is defined as* $\text{windowExpiration}(W_t, BGP) = \min(\{\, t_2 \mid \exists \rho, \mu, t_1 : \rho \in BGP \wedge \mu[BGP] \subset W_t \wedge \text{nextExpiration}(\mu[\rho], t, S_{TPF\text{-}QS}) = t_1 \,\})$ *with BGP a basic graph pattern, μ a solution mapping for BGP, $\mu[BGP]$ and $\mu[\rho]$ the applications of μ to BGP and ρ respectively, and $S_{TPF\text{-}QS}$ the stream this window is applied to.*

Finally, we define the set ET under the *Mapping Expire* strategy as the set of time instants at which time a new window will be requested by the client.

Definition 7. $ET_{ME}(BGP) = \bigcup_{i=0}^{\infty} \{\, t_{i+1} \mid t_i = \text{windowExpiration}(W_{t_i}, BGP) \,\}$, *with BGP a Basic Graph Pattern, t a timestamp and W_{t_i} the window at time t_i.*

This set contains the timestamps at which there will be a data expiration to the given BGP, that causes a new evaluation to only be done when the data of a previous window expired. Additionally, we add the initial timestamp t_0 to ET_{ME}.

Mapping Expire Fallback In case of the *Mapping Expire* strategy, when no time intervals were found for a possibly empty solution mapping, a fallback mechanism is required to determine the next reporting time. The TPF-QS engine will consider the last evaluation time, and add twice its query evaluation duration to calculate a new evaluation time.

4.3. Evaluation Time Instants Example

We assume the use case where a single sensor emits observations at a fixed rate of one observation every two time units. Fig. 1a illustrates this use case: since each value is guaranteed to not change within two discrete time units after an observation, we indicate this time interval as a horizontal line with a length of one time unit on the timeline for each observation. If a query is issued for the (current) observation with the default Mapping Expire policy, the evaluation times will be every two time instants. This is because the client will be able to detect the time interval for each observation, and it will slide the next window to the expiration time of that time interval. Furthermore, the same evaluation times can be achieved by using the Periodic policy with a period of 2.

If we modify our use case where observations arrive more irregularly, but each one still remains unchanged for two time instants, the evaluation times of Fig. 1b will be reached with the Mapping Expire policy. Here we can observe the fallback strategy that is used when the engine calculates an empty result set for an evaluation time as explained in Section 4.2. In this example, we assume that all query evaluations take one time unit, except for the evaluation at time 6, which takes two time units. This means that the fallback expiration time will occur after two time units, except for the expiration time applied after time 6, which will be four time units. Because of this heuristic fallback strategy, the evaluation times results in a missing observation of c.

5. Comparison of Operational Semantics

In this section, we compare the operational semantics of TPF-QS with C-SPARQL, CQELS and Morph_{stream}, as they appear in RSP-QL [11]. Differences and similarities between these three approaches are discussed in the following and summarized in Table 1.

As explained in Section 3, TPF-QS proposes a client-side approach for RDF stream processing that relies on TPF. On the other hand, state-of-the-art RSP engines like C-SPARQL and CQELS work server-side only, exposing APIs for continuous

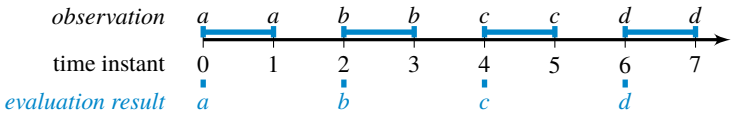

Fig. a: Windows with a default expiration-based window or a fixed window slide parameter of 2.

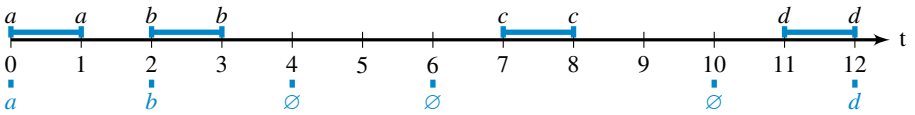

Fig. b: Windows with the default expiration-based window slide parameter, where each query evaluation is assumed to take one time unit, except for the evaluation at time 6, which takes two time units.

Figure 1: Different cases of windows and evaluation times over a sequence of time-annotated observations.

feature	TPF-QS	C-SPARQL	CQELS	Morph$_{stream}$
volatile RDF stream	no	yes	yes	yes
data retrieval	pull	push	push	push
time annotation	time interval	timestamp	timestamp	timestamp
window parameters	t^0 or $\overline{t^0}$, $\overline{\alpha}$, $\overline{\beta}$	$\overline{t^0}$, α and β	$\overline{t^0}$, α and β	$\overline{t^0}$, α and β
RSP-QL dataset	$G_{\mathbb{W}}$ in G_0	$G_{\mathbb{W}}$ in G_0	named $G_{\mathbb{W}}$	$G_{\mathbb{W}}$ in G_0
window policy	ME or P	WC, NC	CC	WC, NC
streaming operators	RStream	RStream	IStream	RStream, IStream and DStream

$\overline{\alpha}$ and $\overline{\beta}$ are fixed to one time unit; $\overline{t^0}$ is an internally defined t^0.
WC=Window Close; NC=Non-empty Content; CC=Content Change.

Table 1: Summary of the main differences between TPF-QS, C-SPARQL and CQELS.

querying. C-SPARQL, CQELS and Morph$_{stream}$ push the results to a required client. TPF-QS instead evaluates the query directly at the client side, pulling stream portions from the server.

This paradigm shift influences how RDF streams are represented: C-SPARQL, CQELS and Morph$_{stream}$ handle an incoming volatile RDF stream on the fly using windows; TPF-QS accesses the RDF stream – that is (possibly temporally) stored with a TPF interface – using time annotations. The user of C-SPARQL, CQELS and Morph$_{stream}$ can control the time dimension specifying window width α and slide β through SPARQL 1.1 extensions with time semantics (see Section 2). TPF-QS hides the time dimension behind the TPF interface, fixing the window width α and β to one time unit. This decision simplifies the query expression, allowing the user to write traditional SPARQL 1.1 queries. While C-SPARQL, CQELS and Morph$_{stream}$ internally define the start time t^0 of a window, TPF-QS optionally allows this to be set to any moment in time.

Similar to C-SPARQL and Morph$_{stream}$, TPF-QS adds the time-varying graph to the default graph, while CQELS enables query-level access using a named time-varying graph.

The TPF-QS window policy to determine evaluation times works in two alternative modes: (a) Mapping Expire (default), where evaluation times are based on data expiration time; (b) Periodic, where the periodicity can be configured to any amount of time units. TPF-QS uses the RStream operator to report results. C-SPARQL reports on "Window Close" with RStream operator too, but it does not show empty results; CQELS uses the IStream operator and reports results on "Content Change". Morph$_{stream}$ reports on "Window Close" without empty results, and allows RStream, IStream or DStream for reporting.

Finally, it is worth mentioning that ETALIS/EP-SPARQL is the only RSP engine with an interval-based time semantics like TPF-QS (see Section 2). Besides this similarity, the operational semantics of the two systems are not comparable under the RSP-QL model: EP-SPARQL does not provide any windowing approach, but CEP operators; TPF-QS has no notion of events at this stage of development.

6. Evaluation

In order to evaluate the performance of TPF-QS compared to other RSP engines for equivalent queries, we extended[2] the RSP benchmark CityBench [1] with TPF-QS support. This allows a performance comparison of our solution with C-SPARQL and CQELS. However, we did not compare with Morph$_{stream}$, as it is not supported by CityBench. We measure server load, client load, query result latency, and query completenes with an increasing number of concurrent queries. In this section, we describe the CityBench benchmark, after which we present our experimental setup, results and a discussion.

6.1. CityBench

CityBench is a benchmarking suite for evaluating RSP engines based on sensor data from the city of Aarhus, Denmark. This benchmark is a good candidate, since it evaluates elements of RSP engines that are of importance in realistic real-time scenarios, like result completeness, scalability and query result latency, and it provides queries over slow streams. Result completeness is measured internally as a fraction by comparing result streams with the expected output stream for each query. Latency is measured as the difference in time between inserting an element in the stream and it appearing as a result.

The original CityBench framework supports the C-SPARQL and CQELS engines. Since these are pure *server-side* engines, the benchmark can run on a single server machine. In order to evaluate TPF-QS, we made three significant changes to the framework.

Given that CityBench provides queries in the C-SPARQL and CQELS query syntax, and TPF-QS only accepts SPARQL 1.1 queries, we created a set of SPARQL queries to semantically equivalent results using on the RSP-QL-based formalization from Section 4 and operational engine comparisons from Section 5, so that they result in an evaluation frequency of 10 seconds. This frequency was chosen because earlier experiments have shown that TPF-QS works best with this order of frequencies or slower. Additional details on how these queries were semantically transformed can be found in the appendix[3]. We selected 6 of the 12 queries as the others used SPARQL filter operators that are currently unsupported by the TPF-QS engine.

In order to ensure the server and client load were measured independently, TPF-QS clients ran on different machines, separate from the server. All clients connected to the server through a cache, which was flushed when data changed in the dataset. We limited the amount of clients running on a single machine to the number of cores machine cores.

During experiment initialization, CityBench sent queries to each client machine and monitored its results. As with the server, the load of these machines was measured for the duration of the experiment.

6.2. Experimental Setup

Our experimental setup consisted of one server machine and eight client machines. The client machines were only used to run TPF-QS clients. while C-SPARQL and CQELS only ran on the server. Our experiment was executed on compute-optimized c3.2xlarge Amazon EC2 machines, with eight High Frequency Intel Xeon E5-2680 v2 (Ivy Bridge) Processor cores and 15 GiB of memory. All experimental results together with instructions to repeat them can be found at https://github.com/LinkedDataFragments/CityBench-Amazon.

We ran the experiments with the following full-factorial setup:

- Engine: TPF-QS, C-SPARQL, CQELS
- Query: Q1, Q2, Q3, Q6, Q9, Q11[2]
- Number of concurrent clients: 1, 16, 32, 64

The experiments were run with a minimum duration of 15 minutes. Every five seconds, we measured server CPU, query result latency and result completeness. For TPF-QS, we also measured bandwidth and client CPU usage in order to see the effects of moving load to the client. We implemented an extension of the TPF server with quad support[4] in order to expose our RDF streams using graph-based annotation with expiration times [23].

For each query, we consider three null hypotheses stating that server CPU, query result latencies and result completeness are equal across the three query engines. We tested these hypotheses for equality using the Kruskal-Wallis test, and

[2] Modified version of CityBench and queries: https://github.com/LinkedDataFragments/CityBench
[3] Appendix describing the used queries: https://zenodo.org/record/200844#.WFEMB8MrKHp
[4] Implementation of the extended TPF server: https://github.com/LinkedDataFragments/Server.js/tree/feature-querystreamer-support

performed post-hoc analysis with the Nemenyi test for pairwise comparisons. These results of these tests are summarized in Table 2 and will be further discussed hereafter.

6.3. Concurrency

Fig. 2 shows the evolution of server CPU usage for the three evaluated engines. The load in the case of TPF-QS is significantly lower or equal for 3 of the 6 queries when compared to CQELS, and for 5 of the 6 queries when compared to C-SPARQL, as validated by comparison of their means in Table 2. For the other queries, the load with TPF-QS is higher. For each query, we can therefore reject the null hypothesis that server load is equal for the three query engines.

In Fig. 3 we can see the average client CPU usage of TPF-QS for all queries, which illustrates the new trade-off in server and client load with the TPF-QS approach. For most queries, we see an initial peak in client load, after which this drops to a significantly lower CPU usage for all queries except for Q2 and Q9. This initial peak corresponds to the initial query rewriting phase, which was discussed in previous work [23].

Fig. 4 shows the corresponding query result latencies for an increasing amount of concurrent queries. Table 2 confirms that these latencies are significantly lower or equal for 3 of the 6 of the queries when compared to CQELS, and for 4 of the 6 queries when compared to C-SPARQL, For each query, we reject the null hypothesis for equal latencies across the three query engines. The queries with a lower latency for TPF-QS also have a relatively low server CPU usage.

For almost all queries, we can see that in the case of TPF-QS, the latencies decrease until a certain number of concurrent clients is reached, after which the latencies increase. This effect is very clear for Q6, where the latency eventually becomes higher than that of C-SPARQL. For Q9 we see a different story, where eventually C-SPARQL's initially lower latency goes above that of the TPF-QS, which suggests that C-SPARQL does not scale well for this query.

It is clear that none of the engines performs better than the others for all queries. TPF-QS outperforms C-SPARQL and CQELS for simple queries Q1 and Q3, CQELS outperforms the others for the more complex queries Q6, Q9 and Q11, and C-SPARQL is faster for the simple query Q2.

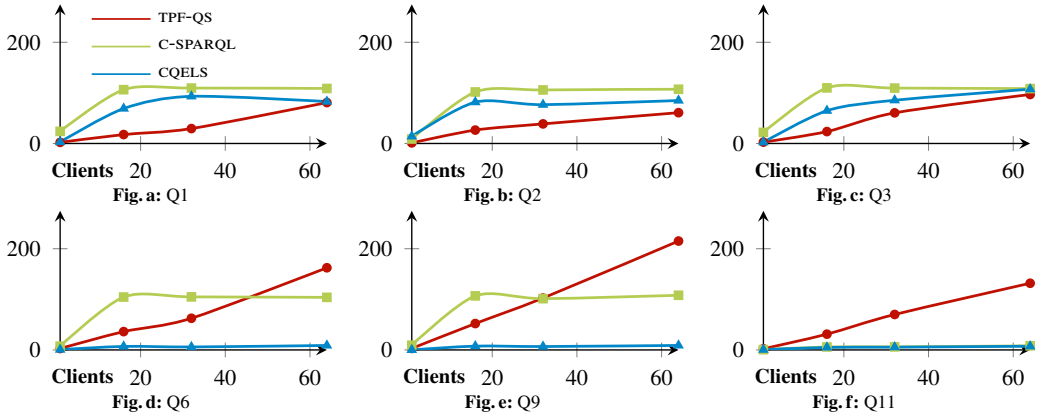

Figure 2: Average server CPU usage (%) for the TPF-QS, C-SPARQL and CQELS is higher for increasing amounts of concurrent queries.

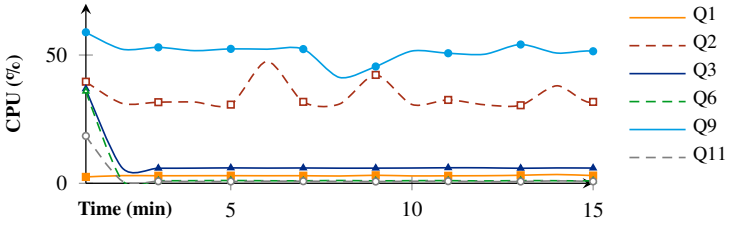

Figure 3: Average client CPU usage of the TPF-QS initially peaks, and then converges to a lower value for most queries (except Q2 and Q9).

6.4. Completeness

Fig. 5 shows the evolution of completeness for the different queries in terms of an increasing number of concurrent clients for the three evaluated engines. When we compare our solution with CQELS, we see that for Q1, TPF-QS has a

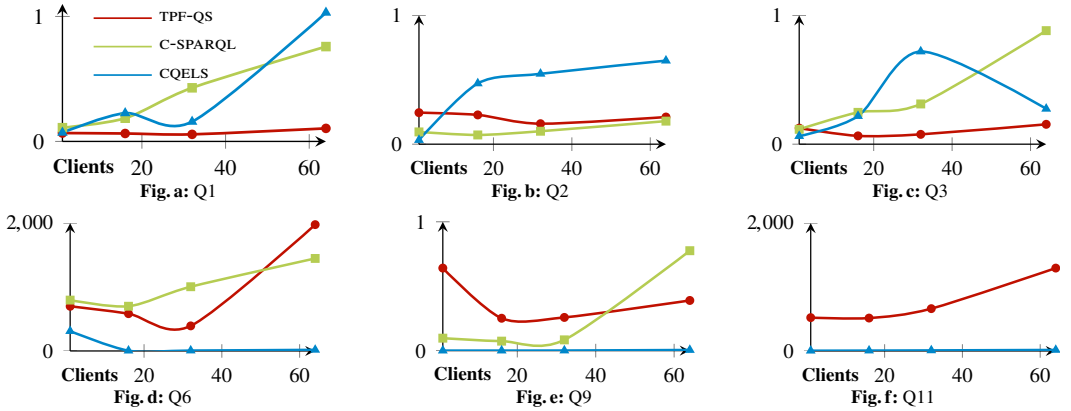

Figure 4: Average latencies (ms) for the TPF-QS, C-SPARQL and CQELS is typically higher for an increasing amount of concurrent queries.

	server CPU comparison				latency comparison				completeness comparison			
	CQELS		C-SPARQL		CQELS		C-SPARQL		CQELS		C-SPARQL	
Q	Δ	*p*-value	Δ	*p*-value	Δ	*p*-value	Δ	*p*-value	Δ	*p*-value	Δ	*p*-value
Q1	**-28.28**	0.0002	**-54.57**	0.0000	**-2 796.21**	0.0000	**-2 979.52**	0.0000	**0.05**	0.0002	**0.02**	* 0.1244
Q2	**-32.55**	0.0004	**-49.08**	0.0000	**-1 656.44**	0.0000	1 473.21	0.0002	-0.64	0.0000	-0.63	0.0000
Q3	**-16.60**	* 0.1685	**-41.56**	0.0000	**-2 110.57**	0.0001	**-2 839.88**	0.0000	-0.09	0.0000	-0.08	0.0000
Q6	60.41	0.0000	**-14.17**	* 0.2725	827.86	0.0000	**-74.20**	* 0.6142	-0.14	0.0000	-0.11	0.0000
Q9	87.39	0.0000	**11.83**	* 0.5643	3 831.19	0.0000	1 273.78	0.0013	-0.18	0.0000	-0.15	0.0000
Q11	54.26	0.0000	53.66	0.0000	739.97	0.0000	**-Inf**	0.0000	-0.09	0.0000	**0.91**	0.0000

Table 2: Difference in means for server CPU (%), latency (ms) and result completeness (fraction) for the TPF-QS with CQELS and C-SPARQL at a frequency of 10 seconds. The *p*-value indicates the significance of his equality; values annotated with a star indicate equality with a confidence level of 95%, Combinations for which the TPF-QS performs equal or better are marked in bold.

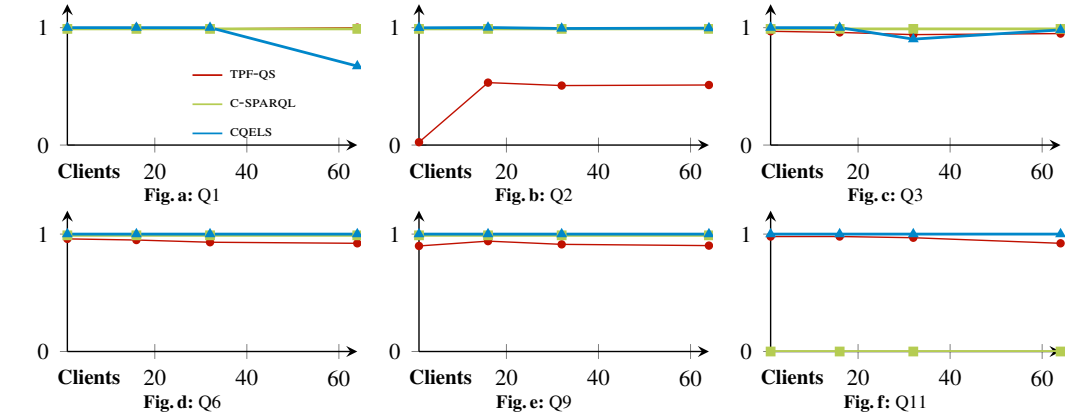

Figure 5: Completeness (fraction) for the TPF-QS, C-SPARQL and CQELS mostly decreases for an increasing amount of concurrent queries.

higher completeness due to the simplicity of the query. This is confirmed by the comparison of their means in Table 2. Compared to C-SPARQL, the completeness level for Q1 is equal, while for Q11 TPF-QS performs much better. For all other queries, TPF-QS has a lower completeness. For each query, we can reject the null hypothesis saying that completeness is the same across the three query engines.

We observe that for all engines, the completeness decreases for an increasing number of concurrent clients, which is caused by the increase in server load. We do however see a different trend for Q2 and Q9 with TPF-QS: it appears that its initially low completeness *improves* with a higher number of clients. This is because of the server-side cache that is able to improve request throughput, which enables the client to download and process a larger fraction of the required data in time before it expires again.

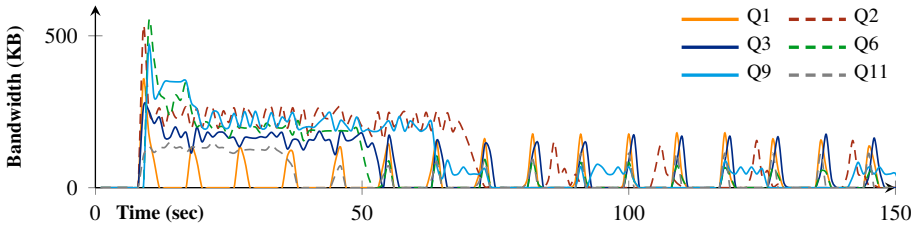

Figure 6: Rhythmic bandwidth usage zoomed in on the first 150 seconds for TPF-QS for all queries with a frequency of 10 seconds without a cache in front of the server. All measurements consist of data collected during one second. Note how the engine does not seem to converge to the right rhythm for Q2 and Q9, confirming the aberrant behavior observed in Fig. 3.

6.5. Bandwidth

Fig. 6 shows the TPF-QS bandwidth usage for all queries, in which the synchronization behaviour of the query engine with the dynamic data can be seen. Verborgh et al. [24] have shown that data transfer is the main bottleneck in the TPF approach. From this figure we can see that for all queries, our server experiences a peak in data transfer at the start of the experiment. This is caused by the initial preprocessing step that needs to be done by the client at the start of each query evaluation. We also observe a highly rhythmic pattern for all queries; the reason for this is that the client re-evaluates the query at the moment the results expire. This expiration time is data-driven, and is defined by the time-annotations on the server, which are fixed at a length of 10 seconds.

The shape of each phase is highly dependent on the type of query. For example, Q1 is a smaller query than Q9 in terms triple pattern count. The amount of triple patterns has an influence on the required data transfer for the TPF client's query evaluation due to its iterative algorithm [24]. The larger query Q9 therefore leads to a higher evaluation time.

Fig. 6 also shows a period of high bandwidth usage right after the initial peak for each query. The length of these periods varies strongly with each query. This period is the time that it takes for TPF-QS to synchronize in phase with the server. It exists because the query engine requires a non-negligible amount of query evaluation time. When certain statements expire during this time, solution mappings become invalid, what will cause the fallback slide parameter to be used for the Mapping Expire strategy. Eventually, most query evaluations get into the correct phase and are synchronized with the data-driven frequency. Q2 and Q9 periodically skip phases because their query evaluation time is too high with respect to this frequency.

6.6. Discussion

We assessed how the query load redistribution to clients affects our solution, and how these effects compare to alternative solutions. We observed that, for simple query types, the server load and query result latency for TPF-QS are both lower than for the alternatives, while for more complex query types the opposite is true. For one simple query, our approach achieved a higher query result completeness. For the others, the completeness was equal or lower, which is caused by the longer query evaluation times. In both cases, there is a significant bandwidth usage.

On the other hand, if observations in data streams arrive at an unpredictable rate, TPF-QS is not the appriopriate solution. This has been illustrated in Section 4.3 where (unexpected) time gaps between subsequent observations, cause the TPF-QS client to not be able to properly derive the next evaluation time.

Results also show an increase in result latencies when a higher number of concurrent clients is reached. This is caused by the cache that is cleared every time new data is inserted into the server. Caching makes TPF requests efficient for static data, but it is more difficult to manage with data streams, which is a point of improvement for future work.

These caching issues will not arise when (static) historical streams are exposed. In fact, TPF-QS is the first engine that combines the worlds of RSP and RDF archiving [15], by allowing a custom t^0 to be chosen, instead of always querying for the current time.

7. Conclusions

In this paper, we formalized the operational semantics of TPF-QS, a technique that combines both server-side RDF stream publication and client-side querying, using the RSP-QL model and we formally and experimentally compared it with C-SPARQL and CQELS.

The results of our evaluation paint an interesting and nuanced story, revealing that there is a potential for lightweight streaming interfaces, even though they do not exhibit optimal behavior in all cases. TPF-QS is effective for simple queries over streams at a low and predictable frequency. Furthermore, this new trade-off in bandwidth and server load is beneficial when bandwidth is cheaper than CPU, such as low-power sensors.

We can compare the trade-off in server and client effort for RSP engines by checking the distribution of their different phases in RDF stream processing [11]: *stream handling*, *windowing of streams*, and *query evaluation*. Typical RSP engines handle everything server-side. TripleWave on the other hand only publishes streams server-side, and requires a separate client-side engine like C-SPARQL to consume their streams. The TPF-QS approach, in contrast, is an all-in-one solution: it requires the server to expose the stream, while the client performs the tasks of windowing and query evaluation. This gives the client more freedom in how to evaluate queries, without being purely dependent on internal query algorithms of server-side engines.

Even though our results show that TPF-QS is not able to solve *all* of the current RSP scalability problems, for simple queries over slow streams, our approach scales better than others for an increasing number of clients. TPF-QS is therefore a next step towards the publication and consumption for low-velocity continuous Linked Data at Web scale. In future work, we plan improvements to our query algorithm, caching strategies and fragmented data publication. Furthermore, we intend to generalize the TPF-QS approach by integrating it within the modular Comunica query platform [22].

References

[1] Ali, M.I., Gao, F., Mileo, A.: CityBench: A configurable benchmark to evaluate RSP engines using smart city datasets. In: The Semantic Web – ISWC 2015 (2015)

[2] Anicic, D., Fodor, P., Rudolph, S., Stojanovic, N.: EP-SPARQL: a unified language for event processing and stream reasoning. In: Proceedings of the 20th international conference on World wide web (2011)

[3] Arasu, A., Babu, S., Widom, J.: The CQL continuous query language: semantic foundations and query execution. The VLDB Journal 15(2) (2006)

[4] Babcock, B., Babu, S., Datar, M., Motwani, R., Widom, J.: Models and issues in data stream systems. In: Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems (2002)

[5] Barbieri, D.F., Della Valle, E.: A proposal for publishing data streams as Linked Data. In: Linked Data on the Web Workshop (2010)

[6] Barbieri, D.F., Braga, D., Ceri, S., Valle, E.D., Grossniklaus, M.: Querying RDF streams with C-SPARQL. SIGMOD Rec. 39(1) (Sep 2010)

[7] Bohlen, M.H., Busatto, R., Jensen, C.S.: Point-versus interval-based temporal data models. In: Data Engineering, 1998. Proceedings., 14th International Conference on (1998)

[8] Buil-Aranda, C., Hogan, A., Umbrich, J., Vandenbussche, P.Y.: SPARQL web-querying infrastructure: Ready for action? In: The Semantic Web–ISWC 2013 (2013)

[9] Calbimonte, J.P., Corcho, O., Gray, A.J.: Enabling ontology-based access to streaming data sources. In: International Semantic Web Conference (2010)

[10] Della Valle, E., Ceri, S., van Harmelen, F., Fensel, D.: It's a streaming world! Reasoning upon rapidly changing information. Intelligent Systems, IEEE 24(6) (Nov 2009)

[11] Dell'Aglio, D., Della Valle, E., Calbimonte, J.P., Corcho, O.: RSP-QL semantics: a unifying query model to explain heterogeneity of RDF stream processing systems. International Journal on Semantic Web and Information Systems 10(4) (2014)

[12] Dell'Aglio, D., Della Valle, E., van Harmelen, F., Bernstein, A.: Stream reasoning: A survey and outlook 1, 59–83 (2017), 1-2

[13] Dimou, A., Vander Sande, M., Colpaert, P., Verborgh, R., Mannens, E., Van de Walle, R.: RML: A generic language for integrated RDF mappings of heterogeneous data. In: LDOW (2014)

[14] Feigenbaum, L., Todd Williams, G., Grant Clark, K., Torres, E.: SPARQL 1.1 protocol. Rec., W3C (Mar 2013), http://www.w3.org/TR/2013/REC-sparql11-protocol-20130321/

[15] Fernández, J.D., Polleres, A., Umbrich, J.: Towards efficient archiving of dynamic Linked Open Data. Proc. of DIACHRON (2015)

[16] Gutierrez, C., Hurtado, C.A., Vaisman, A.: Introducing time into RDF. IEEE Transactions on Knowledge and Data Engineering 19(2) (2007)

[17] Harris, S., Seaborne, A., Prud'hommeaux, E.: SPARQL 1.1 query language. Recommendation, W3C (Mar 2013), http://www.w3.org/TR/2013/REC-sparql11-query-20130321/

[18] Le-Phuoc, D., Dao-Tran, M., Parreira, J.X., Hauswirth, M.: A native and adaptive approach for unified processing of linked streams and Linked Data. In: ISWC 2011 (2011)

[19] Luckham, D.: The power of events: An introduction to complex event processing in distributed enterprise systems. In: International Workshop on Rules and Rule Markup Languages for the Semantic Web (2008)

[20] Mauri, A., Calbimonte, J.P., Dell'Aglio, D., Balduini, M., Brambilla, M., Della Valle, E., Aberer, K.: TripleWave: Spreading RDF streams on the Web. In: International Semantic Web Conference (2016)

[21] Taelman, R., Heyvaert, P., Verborgh, R., Mannens, E.: Querying dynamic datasources with continuously mapped sensor data. In: Proceedings of the 15th International Semantic Web Conference: Posters and Demos (Oct 2016)

[22] Taelman, R., Van Herwegen, J., Vander Sande, M., Verborgh, R.: Comunica: a modular sparql query engine for the web. In: Proceedings of the 17th International Semantic Web Conference (Oct 2018)

[23] Taelman, R., Verborgh, R., Colpaert, P., Mannens, E.: Continuous client-side query evaluation over dynamic Linked Data. In: The Semantic Web: ESWC 2016 Satellite Events (May 2016)

[24] Verborgh, R., Vander Sande, M., Hartig, O., Van Herwegen, J., De Vocht, L., De Meester, B., Haesendonck, G., Colpaert, P.: Triple Pattern Fragments: a low-cost knowledge graph interface for the Web. Journal of Web Semantics 37–38 (Mar 2016)