UNIVERSITY OF TARTU
FACULTY OF SCIENCE AND TECHNOLOGY
Institute Of Computer Science
Infotechnology curriculum

Mikk-Erik Bachmann

Filtering Real-Time Linked Data Streams

Bachelor's thesis (6 ESTS)

Supervisor: Peep Küngas, PhD

Tartu 2016

Filtering Real-Time Linked Data Streams

Abstract:

The amount of linked data in the Web has increased rapidly in recent years. Linked data, often encoded in RDF, is considered as five-star data in the context of open data due to its usability and potential. Although there has been progress in development of linked data technologies and data processing models, still the full potential of linked data has not been realized. One of the challenges is reasoning over linked data streams, which has just recently gained momentum in research. As a result query languages, such as C-SPARQL, have been proposed and corresponding stream reasoning engines have been implemented. However, such implementations have been evaluated so far mostly in academic settings. This work describes a fully functional proof of concept implementation of a stream reasoning system for message-oriented systems, which is capable of exposing a message queue as a linked data stream, which can be filtered by using C-SPARQL - one of the earliest linked data processing engines. The performance of the C-SPARQL engine, which lies at the heart of the implementation, is evaluated by using CityBench benchmark with settings of an enterprise-scale real-time economy application Inforegister NOW!, which is currently under development.

Keywords: RDF, C-SPARQL, REST, Message-oriented Middleware, RabbitMQ, RabbitHub, CityBench

CERCS: P170

Lingitud andmevoogude filtreerimine reaalajas

Lühikokkuvõte:

Viimastel aastetel on Veebis kiiresti kasvanud lingitud andmete hulk. Lingitud andmeid, mis on tihti kodeeritud RDF formaadis, peetakse “viie tärni” andmeteks avatud andmete kontekstis tänu nende kasutatavusele ja potentsiaalile. Kuigi on märgata progressi lingitud andmete tehnoloogiate arengus ja nende töötlemises, pole veel suudetud nende täit potentsiaali saavutada. Üks väljakutsetest on lingitud andmevoogude peal järelduste tegemine, mis on alles hiljuti hakkanud uuringutes koguma hoogu. Nende tulemusena on pakutud välja päringu keeled nagu C-SPARQL ja loodud tuletusmootorite implementatsioonid. Aga neid mootoreid on senini testitud ainult akadeemilistes keskkondades. Selle töö eesmärk on luua täielikult töötav prototüüp lingitud andmevoogude töötlemiseks sõnumipõhistes süsteemides, mis suudab lingitud andmetest koosnevat sõnumite järjekorda näha kui andmevoogu ja filtreerida seda C-SPARQL-i mootoriga, mis on üks esimesi omalaadseid. Selle süsteemi südames olevat C-SPARQL-i mootorit testisime CityBench võrdlusuuringu programmiga võttes arvesse ärivaldkonda kuuluvat reaalaja rakendust Inforegister NOW!, mis on veel arendusfaasis.

Võtmesõnad: RDF, C-SPARQL, REST, sõnum-orienteeritud vahevara, RabbitMQ, RabbitHub, CityBench

CERCS: P170

Contents

1. Introduction

In 2001 Tim Berners-Lee proposed an idea for Semantic Web as the next logical evolutionary step for the World Wide Web, where not only documents, but data as well are linked together.[1] In Semantic Web not only documents but data itself is interconnected and in a format that is machine-readable. This way applications can make inferences and choices themselves based on data they receive.
Although slow at first, the adoption of Semantic Web technologies has accelerated in recent years. According to Schema.org, whose mission is to create, maintain, and promote schemas for structured data on the Internet, 10 million webpages use their markup on their websites and email messages. Many big companies such as Wikipedia and Facebook expose their data in a structured form.
But in real-world applications data can also take the form of a stream where data is infinitely generated in big quantities and they lose their usefulness fast. Such is the case for soft real-time systems where incoming info has to be processed swiftly for the system to run optimally. The field of linked data streams is still evolving and its technologies have not been adopted yet in commercial applications.

The aim of this thesis is to create a proof of concept implementation for filtering real-time linked data in a real world application using C-SPARQL (continuous SPARQL), REST(representational state transfer) and MOM (message-oriented middleware) technologies.
We also run different tests on the C-SPARQL engine to measure its performance under different loads using a modified version of CityBench benchmark. Test parameters and queries were chosen to match the characteristics of Inforegister NOW! mobile application. The application uses a linked stream API (Application Programming Interface) to leverage data for real-time decision-making.[2]

The rest of the paper is structured as follows: in Section 2 we give a brief overview of the related technologies at the core of this solution whose knowledge is needed to understand the rest of this document; in Section 3 a short list of related work is presented; Section 4 introduces the concrete technologies used and in the second part detailed descriptions of proof of concepts is given; in

Section 5 load testing results of the C-SPARQL engine are shown and in the final Section 6 concludes the thesis and discusses possibilities of future work.

2. Background

2.1 RDF

This so called Linked Data is made possible by a common data presentation language RDF (Resource Description Framework). In the World Wide Web documents of data (i.e. web pages) are linked together and identified by URL-s (Uniform Resource Locator). In RDF this idea is taken further by giving similar identifiers to data itself called URI-s (Uniform Resource Identifier), which is a superset of URL. In RDF data is grouped together into subject-predicate-object triples called *statements*, where *subject* is the described resource, *predicate* is resource's property being described and *object* is the property's value. All three parts of these triples are usually represented by URI-s (although object or the value can also be a constant value called a *literal)*. This way objects and predicates in one statement can be a subject in another forming a directed graph where data is linked together and both data itself and relationships between data are identified and described.[3]

RDF has several well known serialization formats like RDF-XML, JSON-LD, Turtle and N-TRIPLE. Figure 1 shows a RDF graph example while Figure 2 shows the same graph represented in N-TRIPLE format with fictional URIs.

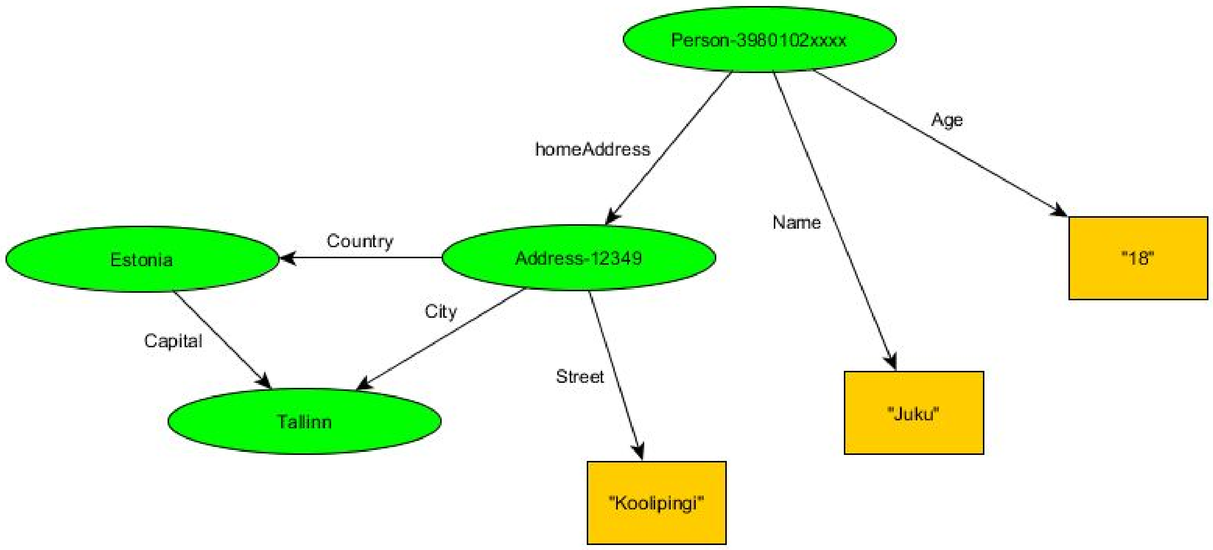


Figure 1: Sample RDF graph.

```
<http://www.example.org/person#3980102xxxx> <http://www.example.org/person#homeAddress>
<http://www.example.org/address#12349> .
<http://www.example.org/person#3980102xxxx> <http://www.example.org/person#age> "18" .
<http://www.example.org/person#3980102xxxx> <http://www.example.org/person#name> "Juku" .
<http://www.example.org/address#12349> <http://www.example.org/address#city>
<http://www.ex.org/cities/Tallinn> .
<http://www.example.org/address#12349> <http://www.example.org/address#street> "Koolipingi" .
<http://www.example.org/address#12349> <http://www.example.org/address#country>
<http://www.ex.org/countries/Estonia> .
<http://www.ex.org/countries/Estonia> <http://www.ex.org/country/capital>
<http://www.ex.org/cities/Tallinn> .
```

Figure 2: Sample RDF graph in N-TRIPLE format.

2.2 SPARQL

SPARQL(SPARQL Protocol and RDF Query Language) is similar to other query languages like SQL in way that it also has same operations like SELECT, UPDATE, DELETE, ORDER BY, GROUP BY etc, but instead of querying over relational data in database tables, it works on RDF data stores. It uses pattern matching to find triples. In the WHERE clause one or more parts of the subject-predicate-object triples is replaced by a variable. All triples are found, that match to the concrete triple parts, and the rest of the parts are bound to the corresponding variables as values.[4] A sample query is given in Figure 3. In plain english this could mean “Find all the streets in Tallinn, where a person named Juku lives”. In Figure 4 a sample result is shown, if this query would be run on the sample graph in Figure 1 and 2.

```
SELECT ?street
WHERE {
 ?person <http://www.example.org/person#name> "Juku" .
 ?person <http://www.example.org/person#homeAddress> ?address .
 ?address <http://www.example.org/address#street> ?street .
 ?address <http://www.example.org/address#city> <http://www.ex.org/cities/Tallinn> .
}
```

Figure 3: sample SPARQL query and a sample result.

street
“Koolipingi”

Figure 4: result of a query in Figure 3, when run on graph in Figure 1.

2.3 RESTful web services

REST(representational state transfer) is a software architecture style that imposes certain constraints on a system. When applied correctly, these constraints help achieve desired non-functional requirements like scalability and modifiability, which help the software work better. One of the more notable constraints is *uniform interface*, which simplifies and decouples different system components, so that they can be developed separately. RESTful web services accomplish this by having an API which exposes resources through URI-s. Standard HTTP methods GET, PUT, POST and DELETE are called against this URI-s for retrieve, create, change and delete operations. The client making the HTTP requests and the server never pass the resource itself to each other, but its representational state, which is in a uniformly agreed upon format like JSON or XML.[5]

2.4 Message-oriented middleware

Another trend in web applications is the adoption of message-oriented middlewares (MOM) (a.k.a. message broker). MOM is a software or hardware used in modular and distributed systems, that mediates communication between other components. This allows for loosely coupled easily scalable systems. Sender (a.k.a. publisher or producer) does not need to know about the location or nature of the receivers (a.k.a. consumers) and vica versa. Both new receivers and senders can be added with little effort. Message-oriented middleware also provide an interface for administration, which enables monitoring and tuning the messaging. Messages are usually sent asynchronously: after sending all the messages the sender can continue with other activities while the messages wait in the middlewares queues for receivers to consume thus removing the delay of waiting for the response for the sender.[6]

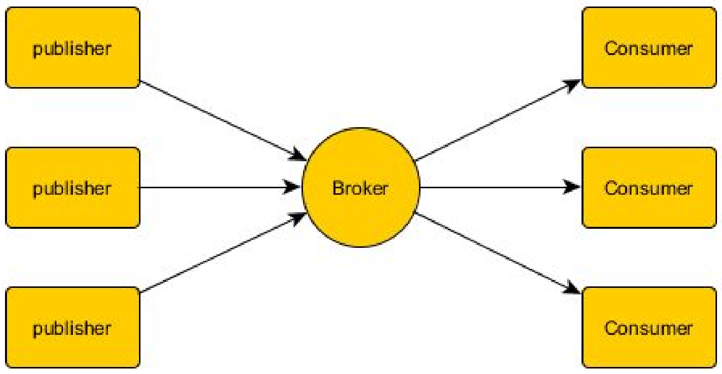


Figure 5: simplified architecture of a system with message-oriented middleware.

3. Related work

Although the field of RDF stream processing(RSP) is still relatively young, several other solutions have been proposed. CQELS is another RSP engine, that uses a native approach instead delegating the processing to other systems.[7] ETALIS is an event processing system that detects complex events from a stream of atomic events combined with static background knowledge. It uses another extension of SPARQL called EP-SPARQL(Event Processing SPARQL).[8] One more SPARQL extension called SPARQLstream aims to provide ontology-based access to relational data by transforming SPARQLstream's SPARQL-like queries into continuous query language SNEEql and later transforming the results into RDF triples.[9] SparkWave is a solution for doing continuous pattern matching over RDF streams using pattern matching algorithm called Rete.[10] INSTANS is another event-processing platform also based on Rete algorithm.[11]

For these RSP engines several benchmarks have also been developed. The first two benchmark for linked data stream engines that emerged were SRBench and LSBench. SRBench tested the functionality of the engines[12] while LSBench also tested correctness based on the number of output elements, performance and scalability tests.[13] CSRBench was an upgrade to the SRBench that concentrated on the correctness of the results of the queries.[14] Older benchmark results are less relevant as the engines, they tested, are being developed consistently. In that regard CityBench and YABench are newer benchmarks that build upon the previous ones. YABench aims to combine and extend previous SRBench, LSBench and CSRBench benchmarks.[15] But at the time of writing this thesis, it is not suited to run tests with multiple queries, which is one of our requirements.

Considering C-SPARQL is still in each earlier stages of development, it has not found much use yet in nonacademic applications. One such is a part of a project called ModaClouds - MOdel-Driven Approach of design and execution of applications on multiple Clouds. One of this project's deliverables is a run-time environment for multi-cloud applications. One of its components is a monitoring platform. The data being monitored is in RDF format and C-SPARQL engine is used to filter it for data visualization or detect on-the-fly patterns to make changes to the system for better quality of service. The C-SPARQL engines RESTful interface is also used.[16]

WAVES is a project whose objective is to deploy a real-time semantic stream management platform for smart urban technologies. One of its main components is a reasoning system for RDF streams. For that they have run some experiments on CQELS and C-SPARQL engines measuring execution time and memory consumption with different RDF triple rates, window sizes, number of streams and static data size.[17]

In this [18] paper a query engine is proposed for large throughput of sensor data and large volumes of already stored data. This solution does not work with RDF streams, but instead large quantities of static data. They ran experiments on their engine to measure query performance. One of the experiments they ran multiple queries in parallel from 10 to 1000 at the same time.

4. Implementation

In the first part of this Section we explain each component separately and in the second part we present two proof of concepts where they are combined into a cohesive system.

4.1 C-SPARQL

C-SPARQL(Continuous-SPARQL) is an extension of SPARQL. It enables running SPARQL queries continuously on RDF streams. It achieves this by adding a timestamp to RDF triples, when they are added to the stream creating a new data structure for RDF Streams: an ordered list of triple-timestamp pairs. The window clause in this query language makes it possible to filter and compute results on specific subset of the stream after regular time intervals. It has two parts: a range and a step. The range is the size of the window and step is a frequency of execution for the query. So for example a range of 10 seconds and a step of 5 seconds means that the query is executed every 5 seconds on triples whose timestamp is not older than 10 seconds. C-SPARQL also adds a FROM STREAM keyword for identifying the streams of RDF triples. An example of C-SPARQL query is shown in Figure 6. This query finds every organization where a person working for organisation 11215399 is a member of.

```
REGISTER QUERY exampleQuery AS
SELECT ?person ?lead
   FROM STREAM <http://ex.org/test#memberships> [RANGE 10s STEP 5s]
WHERE {
 <https://graph.ir.ee/organizations/ee-11215399> <http://www.w3.org/ns/org#hasMember> ?person .
 ?lead <http://www.w3.org/ns/org#hasMember> ?person .
}
```

Figure 6: C-SPARQL query example.

C-SPARQL also allows having multiple FROM STREAM clauses for querying over several streams at once. Adding static RDF stores as a source is also possible with a FROM keyword followed by URL to a RDF graph. A timestamp function can be used to get the timestamp of a stream element e.g. to find out which triple arrived first. The results of a stream can be a set of bindings in case of a REGISTER QUERY and SELECT keywords, but also a new stream, if REGISTER STREAM and CONSTRUCT keywords are used instead. In this case the new stream can also be an input to another C-SPARQL query.[19]

4.2 C-SPARQL Engine

C-SPARQL engine is a RDF stream processing tool written in Java. It lets users register *streams*, *queries* and *observers*. Stream objects are logical representations of the linked data streams, which can be *feed* with RDF triples. Queries are C-SPARQL queries, which are registered to run on one or more streams. Observers are registered on the queries to observe the results computed by the queries.

The engine has two main components - a Data Stream Management System (DSMS) for the stream related parts of the query and SPARQL reasoner for the static part. As new data comes into the engine, the DSMS partitions it according to the window clause and gives the parts to the SPARQL reasoner. The reasoner runs the static part of the C-SPARQL query on the data and returns the results back to the DSMS which outputs a new stream of relational data or RDF data depending on the keyword at the beginning of the query.[20] In our version of the engine, the used DSMS implementation is Esper and the SPARQL reasoner is Jena.

4.3 C-SPARQL RESTful interface

C-SPARQL RESTful interface exposes C-SPARQL engines stream, query and observer operations with a REST API. Figure 7 shows an illustration of the engine with a RESTful interface. All the relevant REST calls are brought in Figure 8. [16]

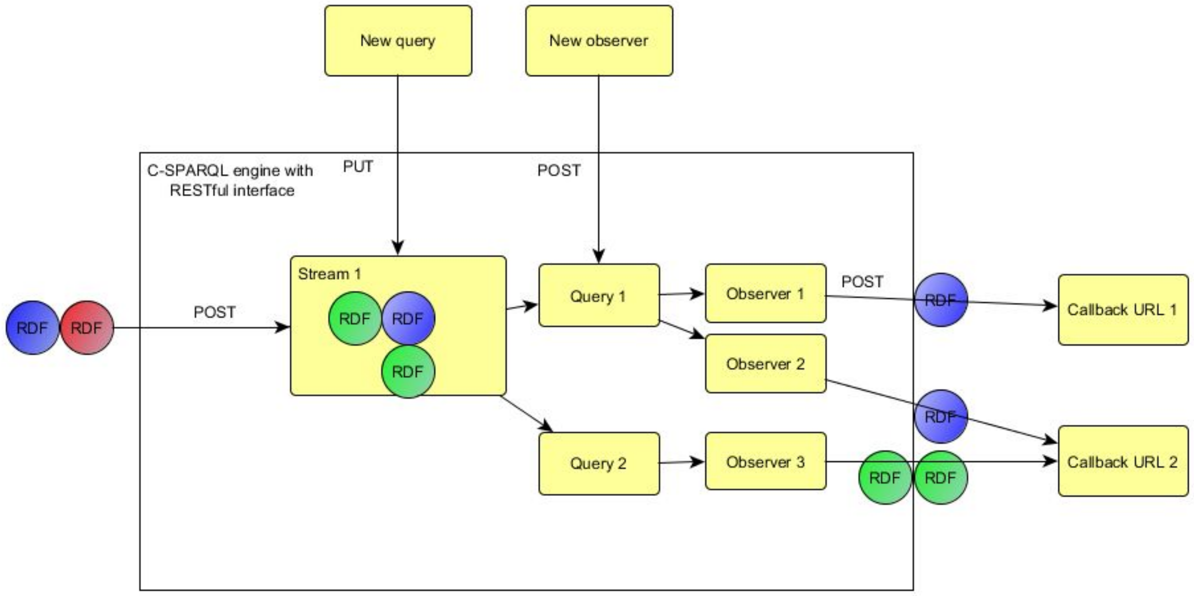


Figure 7: illustration of C-SPARQL engine with a RESTful interface.

Method	**Address**	**Body**	**Description**
RDF Streams			
PUT	/streams/<id>		Register new stream
DELETE	/streams/<id>		Delete specified stream
POST	/streams/<id>	RDF model	Stream(feed) new information
GET	/streams		Get the list of streams
C-SPARQL queries			
PUT	/queries/<id>	query	Register new query
DELETE	/queries/<id>		Delete specified query
POST	/queries/<id>	callback URL	Add new observer
POST	/queries/<id>	action=<pause|restart>	Change query status
GET	/queries		Get the list of queries
Observers			
DELETE	/queries/<id>/observers/<obsId>		Delete specified observer
GET	/queries/<id>/observers		Get the list of observers

Figure 8: C-SPARQL engines RESTful interfaces method descriptions

4.4 RabbitMQ

RabbitMQ is a versatile message-oriented middleware application written in Erlang. It was written as an implementation of AMQP, but supports other messaging protocols as well via plug-ins. It supports many different languages to publish and consume messages including Java. Two key terms of RabbitMQ model, which is based on the AMQP, are *exchanges* and *queues*. Queue is a buffer for messages. Consumers will listen on the queues. When a new message arrives to the queue, it will be sent to a consumer or stay in the queue until a new consumer starts listening on the queue. Publishers send messages to exchanges. Exchanges job is to forward published messages to corresponding queues according to the type of the exchange and *bindings* between queues and exchanges. Bindings are rules, that exchanges use to know which queues to forward messages to. For example if a message with a routing key “warning” is sent to an exchange with a *direct* type, the exchange will route the messages to all the queues, that are bound with a “warning” binding. If a messages is sent to a *fanout* exchange, then it is forwarded to all the bound queues regardless of the bindings and routing keys. Other exchange types include *topic*, which is a more complex version of the direct type, and *header*, which uses messages headers instead of routing keys for routing.[21]

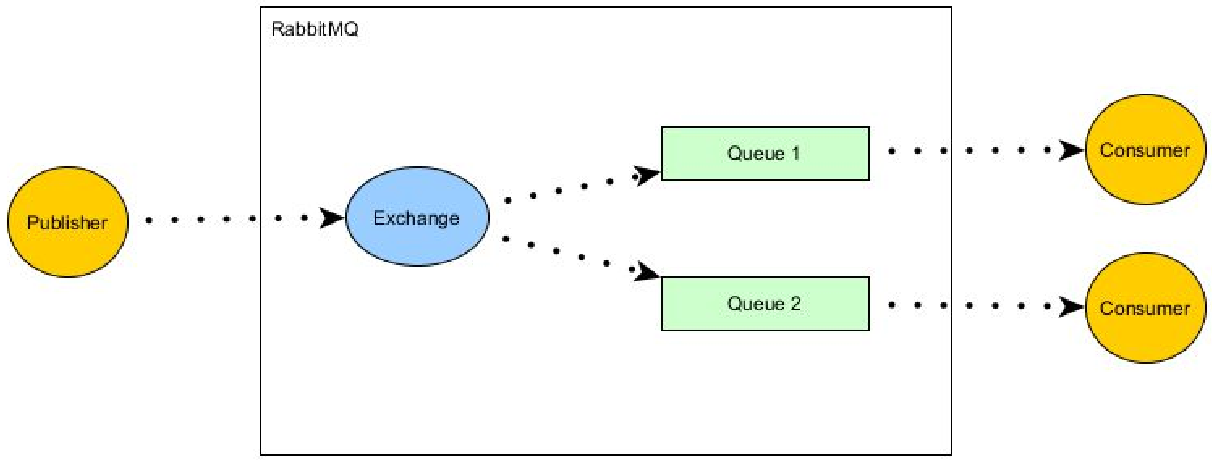


Figure 10: RabbitMQ publishing and consuming model example.

4.4.1 RabbitHub

RabbitHub is a RabbitMQ plugin, which was at first created as a PubSubHubBub protocol implementation for RabbitMQ. PubSubHubBub is a simple publish-subscribe protocol over HTTP. It defines a *publisher*, *subscriber* and a *hub*. A publisher creates a topic on the hub and starts publishing information to it. Subscriber subscribes to a topic on the hub and whenever new info is published on the topic, the hub sends it to all who are subscribed to it. This way users don’t have to poll regularly for new information (pull) but instead hub sends it automatically when it arrives (push) [22]. Because RabbitMQ also supports publish-subscribe pattern, this plug-in has in effect become a RESTful interface for manipulating RabbitMQ exchanges and queues. But unlike other PubSubHubBub implementations which rely on Atom format, this one is content-agnostic meaning any sort of content can be send to and from the hub.

Method	**Address**	**Body**	**Description**
PUT	/endpoint/q/<id>		Create a new queue
PUT	/endpoint/x/<id>?amqp.exchange_type=<type>		Create a new exchange
DELETE	/endpoint/q/<id>		Delete specified queue
DELETE	/endpoint/x/<id>		Delete specified exchange

POST	/subscribe/q/<id>	hub.mode=subscribe
		hub.callback=<callback url>
		hub.topic=<topic>
		hub.verify=<sync|async>
		hub.lease_seconds=<lease seconds>
			Subscribe to specified queue

POST	/subscribe/x/<id>	Same as above	Subscribe to specified exchange

POST	/subscribe/q/<id>	hub.mode=unsubscribe
		hub.callback=<callback url>
		hub.topic=<topic>
		hub.verify=<sync|async>
		<token>
			Unsubscribe from specified queue

POST	/subscribe/x/<id>	Same as above	Unsubscribe from specified exchange
POST	/endpoint/q/<id>?hub.topic=<topic>	message	Send message to specified queue
POST	/endpoint/x/<id>?hub.topic=<topic>	message	Send message to specified exchange

Figure 11: RabbitHub RESTful interface. [23]

4.5 Helper applications

For our proof on concept a couple of helper applications are also needed: a simple servlet application for registering streams and queries and to receive and show the results of the queries. And secondly an application for generating a RDF stream to run our queries on. Both are implemented in Java.

4.5.1 Web servlets for registering queries and receiving results

This is a lightweight Java servlet-based web application. Java servlets are small programs in a web server whose purpose is to handle clients requests, most commonly HTTP requests. This application has three forms each with its own servlet in the backend(i.e. server): one for registering a query, one for registering a stream and one for subscribing to a RabbitMQ exchange via a RabbitHub plug-in. The stream servlet sends a HTTP PUT request to a C-SPARQL RESTful interface to create a stream. In the same way the query registering servlet sends HTTP requests for registering the query and adding an observer to it (see Figure 6). The observer callback URL points to this application's results servlet This servlet writes the received RDF triples to a file and also handles showing of the results. There is also a servlet for subscribing to a RabbitMQ exchange via RabbitHub again with a callback URL pointing back to this application.

4.5.2 Stream data generator

The stream generator can generate dummy triples in-code, read them from a file or dictionary of files or read them from a RabbitMQ queue. It feeds these triples to a C-SPARQL engine through a Java API for C-SPARQL's RESTful services.

4.6 Two Proof of concepts (PoC)

The repository for the Proof of Concepts is in Appendix 1.

4.6.1 First PoC

This PoC's aim is to show that a C-SPARQL's REST API and RabbitMQ as a message-oriented middleware can be combined to form a simple RDF stream filtering solution, which allows processing RDF streams by HTTP calls. In Figure 9 we can see the architecture of PoC 1. It works as follows:

- It integrates C-SPARQL RESTful interface with RabbitMQ so when a new stream is registered(**1**), a new RabbitMQ exchange is also created with the same name as the stream(**2**). This is achieved by adding a code snippet to the REST API implementation, which creates the exchange on stream registering via RabbitHub plug-in.
- User registers a query and an observer with the query servlet(**3**).
- Data generator sends triples to the engine(**4**).
- Observer registering code is also upgraded to not point directly back to the initial callback (which pointed to the result servlet), but to RabbitHub instead. This way the query results are sent to the RabbitMQ exchange with the query name as the routing key(**5**).
- When a user subscribes to the exchange using the subscribing servlet of the helper application, a queue is created with a hub.topic as the binding(**6**). If the hub.topic is the

same as the registered queries name and the subscription's callback points to the results servlet, we can still receive the results of the initial query(**7**).

This PoC was also used by a fellow researcher in his thesis project for web monitoring [24].

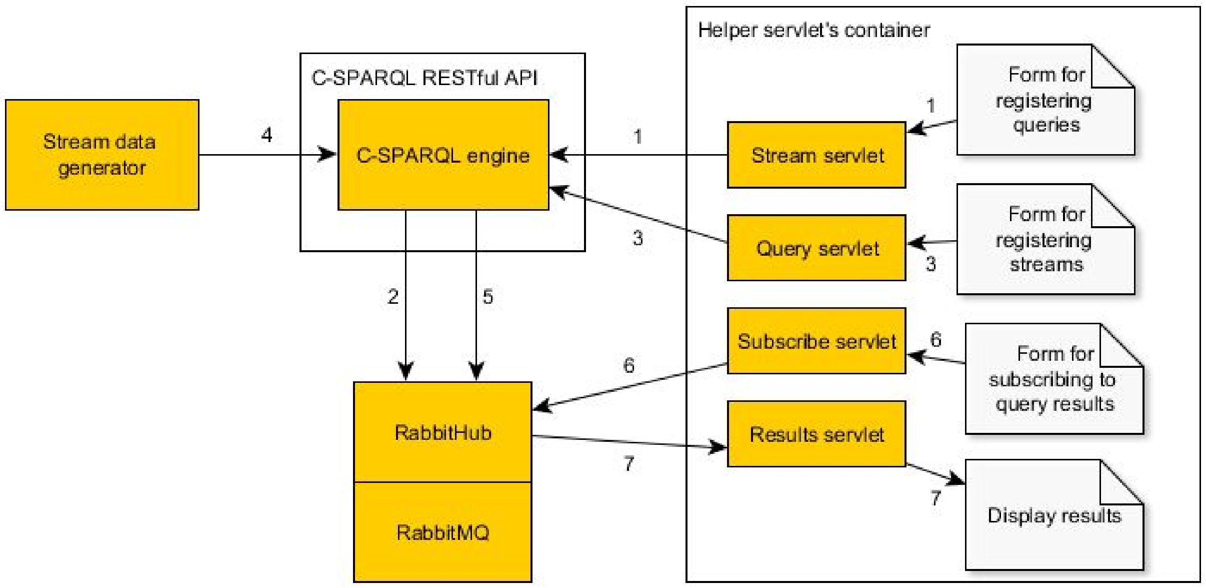


Figure 12: PoC 1 architecture.

4.6.2 Second PoC

Now that we have proven the feasibility of a RESTful RDF stream processing solution, we make another proof of concept for a concrete real world product - Stream API by Register OÜ. The aim is to validate the proposed solution real-life settings. Register exposes data about organizations and its people through its Stream API for businesses so that B2B sales organizations can improve their lead and credit scoring, lead nurturing, advanced analytics etc capabilities. It uses RabbitMQ as its message-oriented middleware.

This time instead of using only HTTP and TCP in a RESTful manner, we use AMQP messaging protocol. For this we add Java code to the data generator, that makes it a consumer for RabbitMQ RDF stream queue. This time we also omit the use of RabbitHub. The RabbitHub, although a versatile add-on, adds an unnecessary layer to the system. Furthermore, as the software is updated less frequently at the writing of this thesis, than the RabbitMQ itself, it might become unreliable with newer RabbitMQ versions. This is an important aspect to consider in a commercial application. The second PoC works like this:

- We register a new stream like before(**1**).
- We start the generator, which starts listening for triples from the RabbitMQ queue. When new RDF triples are added to the queue, the generator receives them(**2**) and forwards them to the C-SPARQL engines stream representation(**3**).
- We register a new query and an observer with a callback to the helper application(**4**).
- As query is run on the stream, results are sent to the result servlet for the user to see(**5**).
- As this version does not have RabbitHub, the subscribe servlet is also not used.

Running this solution we witnessed real results of Inforegister RDF data filtered by the C-SPARQL query.

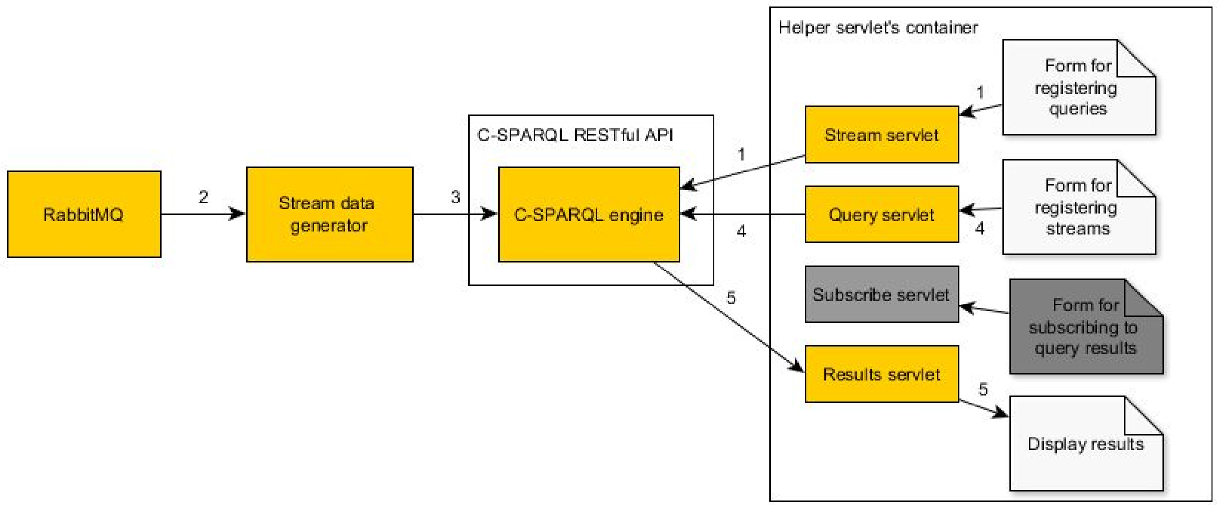


Figure 13: PoC 2 architecture.

4.6.3 Possible solution for the commercial RDF stream filtering like Register Stream API

While RESTful services are good on the edge of the system for interaction with the outside world, it makes less sense to use it internally when a message-oriented middleware is also included. Our Poc2 worked just as well without the RabbitMQ's RESTful interface supplied by RabbitHub. Moreover, it has been found that RabbitMQ's AMQP implementation is better suited for large quantities of data compared to traditional RESTful web services.[25] Having this in mind, one possible solution for a commercial RDF data streaming system like Register Stream API is demonstrated in Figure 14. It provides subsequent possibilities:

- There are pre-made exchanges and queues and C-SPARQL streams already registered on the system, perhaps by an administrator. When a new triple arrives, it is automatically sent to the corresponding stream in the engine(**1**).
- Users can see the streams on the applications webpage and register queries on them(**2**).
- On another page they can also see another the list of registered queries and add observers on them(**3**).
- The results can be sent back to the web applications results page by default(**4**) or to a callback URL over HTTP(**5**) or to a RabbitMQ queue(**6**).
- Users can also register queries and observers via RESTful interface(**7**) and consume both the streams and query results for the RabbitMQ via a another API created for this purpose(**8**).

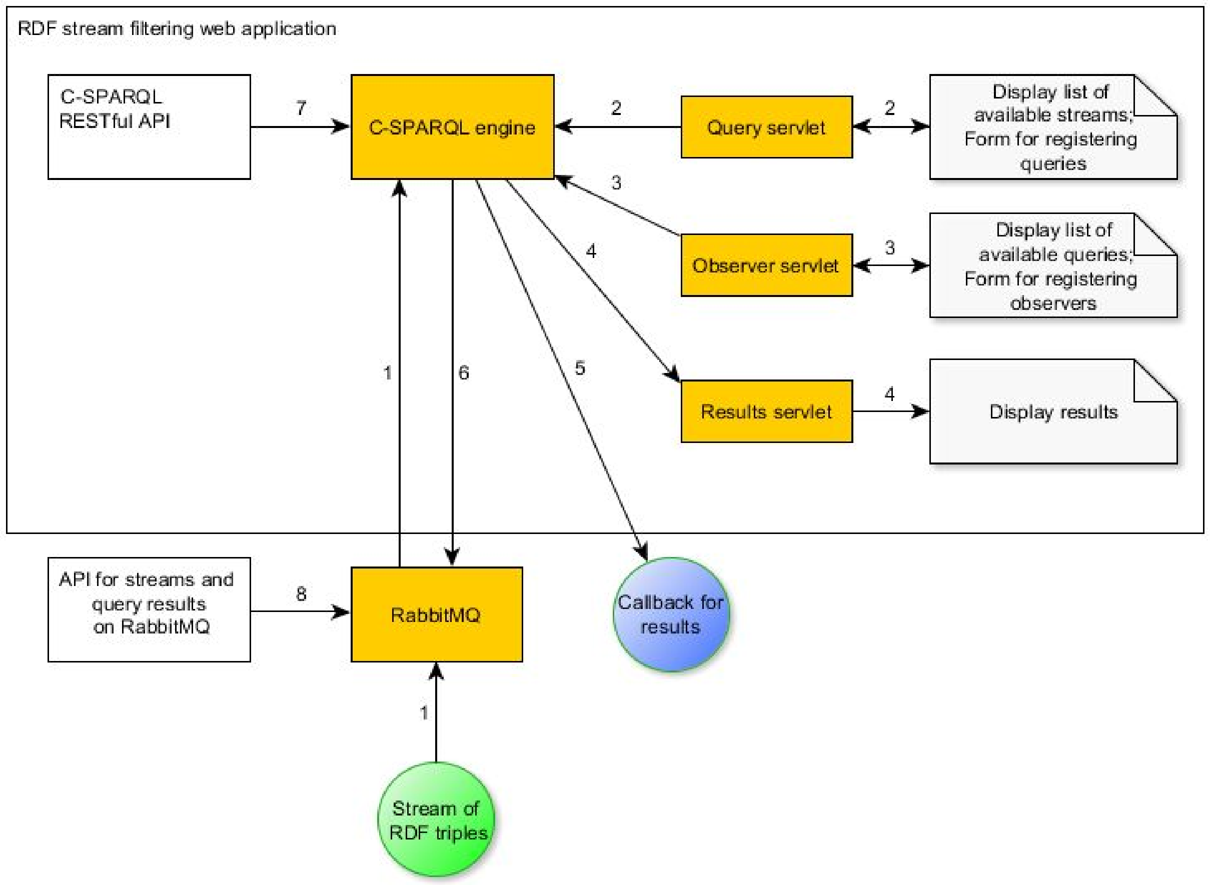

RDF stream filtering web application
C-SPARQL RESTful API
7
C-SPARQL engine
2
Query servlet
2
Display list of available streams; Form for registering queries
3
4
Observer servlet
3
Display list of available queries; Form for registering observers
1
6
5
Results servlet
4
Display results
API for streams and query results on RabbitMQ
8
RabbitMQ
Callback for results
1
Stream of RDF triples

Figure 14: possible RDF streaming solution architecture for commercial infosystem like Inforegister.

This way the system is flexible by allowing multiple ways of registering queries and observers and accessing query results. It has the capabilities of both the simpler widely used REST technology and more complex AMQP, which is better suited for larger datasets. It also lacks redundant components in internal communication - Registering queries and observers using the RESTful interface or web servlets goes straight to the C-SPARQL engine and results go straight to results page, RabbitMQ queue or callback URL depending on the user's preferences.

5. Testing

Since to the best of our knowledge C-SPARQL engine hasn’t found much use in commercial systems, a thorough performance testing is required before it can be used in production environment. In the current Section the results of performance tests are summarized.

5.1 Test case

The queries and dataset for the streams were designed with 2 real-world use cases, both related to real-time lead generation, from Inforegister NOW!, a new product of Register OÜ, in mind. These use cases were represented with 2 different queries, one with a single input stream and another with two streams. The queries are represented in Figure 15 and Figure 16.

```
REGISTER QUERY Q1 AS
SELECT ?obId ?person ?lead
 FROM STREAM <http://ex.org/test#memberships> [RANGE 5s STEP 5s]
WHERE {
 ?obId <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://purl.oclc.org/NET/ssnx/ssn#Observation> .
 <https://graph.ir.ee/organizations/ee-11215399> <http://www.w3.org/ns/org#hasMember> ?person .
 ?lead <http://www.w3.org/ns/org#hasMember> ?person .
}
```

Figure 15: Query 1: Return companies of board members of TENCM OÜ (with reg. no. 11215399) as new leads.

```
REGISTER QUERY Q2 AS
SELECT ?obId ?person ?org ?category
 FROM STREAM <http://ex.org/test#memberships> [RANGE 5s STEP 5s]
 FROM STREAM <http://ex.org/test#organisations> [RANGE 5s STEP 5s]
WHERE {
 ?obId <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://purl.oclc.org/NET/ssnx/ssn#Observation> .
 {
 ?org <http://www.w3.org/ns/org#hasMember> ?person .
 ?org <http://purl.org/goodrelations/v1#category> ?category .
 FILTER regex(?category,'^68')
 }
}
```

Figure 16: Query 2: Return all the persons that work for a company belonging to the real estate category.

First of all, there is a special row at the beginning of the pattern matching WHERE clause in both queries. This is necessary for the CityBench implementation. CityBench measures latency from data's arrival to the engine to the reading of query results by the observer. While most of the triples for the stream are read from a sample stream snapshot file, that is taken from a real Register stream API, there is also a triple, that is added programmatically to every group of triples fed to the stream. It has a unique identifier that is used to identify an observation. This way every group of result triples has an observation id that can be used to determine the observation's time of arrival to the system.

The first query(Q1) is the same as in Figure 6 except the range(window size), which for the base query is same as the step: 5 seconds.

The second query(Q2) adds a stream of organisations to its data sources. It also has a FILTER clause with a regular expression to further filter down the results. It finds all the persons, who work in an organisation that belongs to a category whose code starts with 68. The category codes used in the stream are EMTAK(Eesti Majanduse Tegevusalade Klassifikaator) codes. Codes that start with 68, belong to the real estate related activities.

We measured latency and memory consumption of these queries while varying following parameters:

- RDF triple frequency
- window size
- number of registered queries.

The triple frequency corresponds to how many triples are added to the stream in a second. In a real-world situation the rate in which triples are coming in may change drastically, so a system needs to be robust against higher loads as well. Other times streams don't have enough triples in a small timeframe to be able to make meaningful reasonings on the data, so it would make sense to use a larger window size. Finally we do measurements with different number of registered queries because in commercial systems there might be hundreds or thousands of users who use an application to register similar queries.

5.1.1 Inforegister NOW! use cases

Register queries regularly a pool of remote datasets, including the national registries, Web sites and third-party Web services, to produce streams of RDF data as output. There are two types of output. One is a snapshot of the incoming new data. The other kind are changesets, that show what changed compared to the last snapshot, when it changed and how it changed: whether it is a removal, addition or of new triples. There are 13 different type of data being queried, that is a total of 26 different RDF streams being produced.[2] We also make the following assumptions:

- There will be 200k legal persons (organizations) and 300k board members with a total of 500k subjects to query upon. That means there can be a maximum of 500k different variations of a same kind of query. For example Q1 looks leads on persons, who are a members of organisation (a.k.a. juridical person) with a registration number 11215399. This query could be made for 200k different organisations.
- The system will have 50k users. A user will have an average of 50 queries per stream or an average of 1300 queries total. That makes a total 50k x 1300 = 65 million queries.
- Different streams can have 0.5 - 2 million triples per day.
- A window size can be in the range of 1 month to 1 year.

5.2 Test implementation

Tests were implemented using the CityBench Java application modified to our needs. We chose to use CityBench because it was best suited for our Inforegister NOW! use case. CityBench was created to better simulate real-time applications. The datasets of older benchmarks where static and limited while CityBench's is more dynamic and programmable. At the center of its implementation is what the creators call a *Configurable Testbed Infrastructure* (CTI). Besides the RSP engines, it contains a *Dataset Configurable Module* for configuring streams, *Query*

*Configurable Modul*e for configuring queries and a *Performance Evaluator* for collecting and storing the measurements. Of the several possible parameters, that can be passed to the benchmark, relevant for us are the following: one for setting the stream feeding frequency; one for setting the number of concurrent queries and one for setting the text file from where queries are read.[26]

We ran tests with both queries in three sets, each varying one of the parameters mentioned previously: triple frequency, window size and number of concurrent queries. Each test ran for 10 minutes with fixed configurations. At each minute mark Performance Evaluator saves the arithmetic mean of last minutes latencies and memory usage.

The streams are fed once per second. **96** triples are added to the membership stream: 95 from the snapshot file plus 1 observation triple in code. The same number for the organisations stream is **57**.

Specifications of the computer used for testing are shown in Figure 17. Note that one of the computer components mentioned is RAM, but CityBench is a Java application running on Java Virtual Machine(JVM) that had its maximum memory allocation pool set to 2048 MB.

Component	**Specification**
Operating system	Microsoft Windows 10 Home, 10.0.10586 Build 10586, 64-bit
Processor	Intel(R) Core(TM) i7-4500U CPU @ 1.80GHz, 2401 Mhz, 2 Core(s), 4 Logical Processor(s), x64-based
RAM	8.00 GB
Disk drive	Samsung ssd MZMTE256HMHP-000L1, 256 GB

Figure 17: computer specification used for testing

5.3 Results

For some measurements both graphs with and without high parameter values are included for better clarity.

5.3.1 Varying frequencies

In Figure 18 we can see Q1’s latency and memory usage in relation to different frequencies and in Figure 19 we can see the same for Q2. All these tests had a window size of 5 seconds and 1 query running at a time.

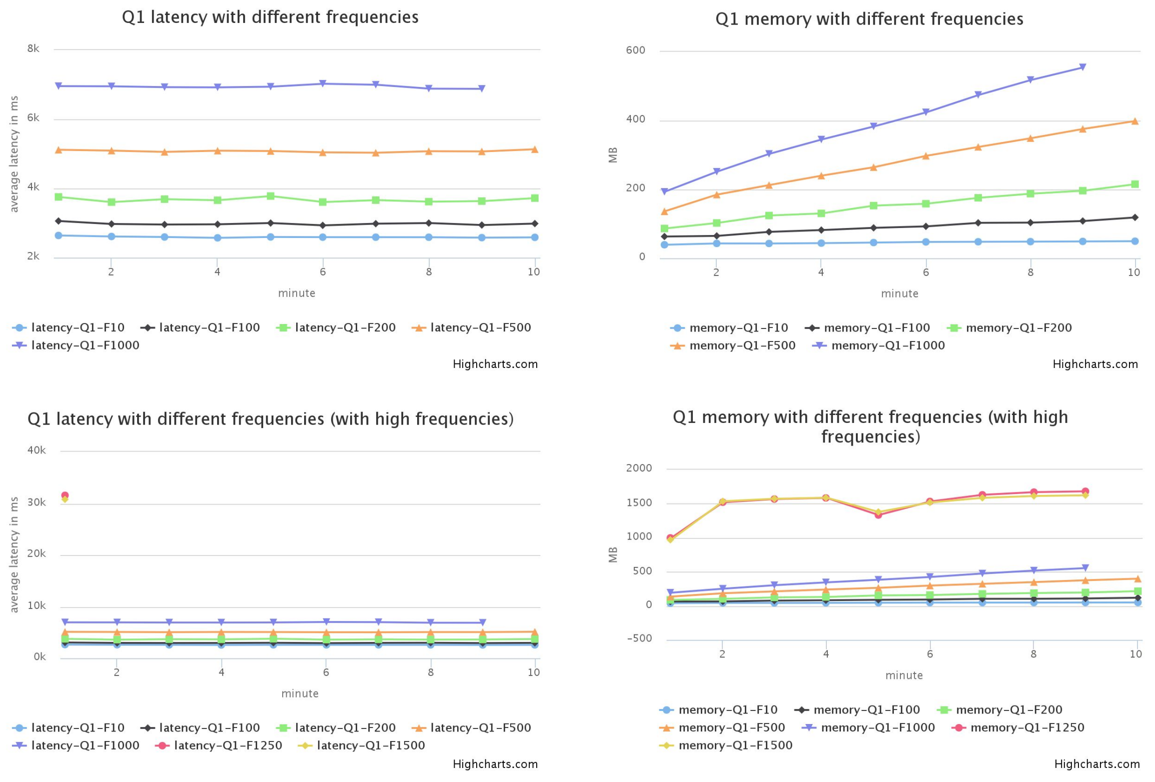
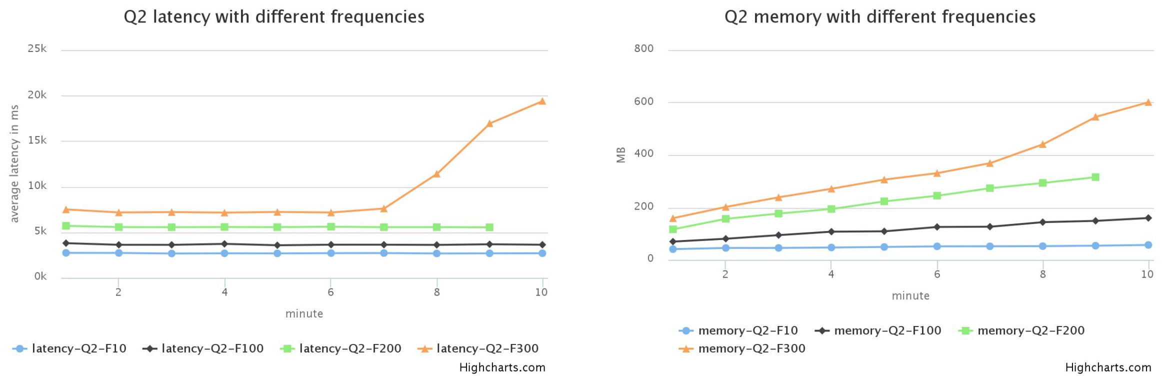
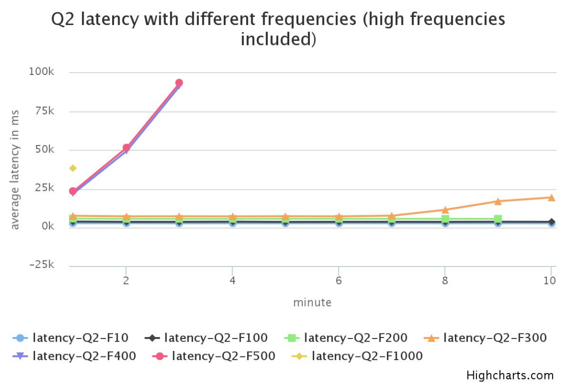


Figure 18: Query 1 latency and memory usage with varying frequencies





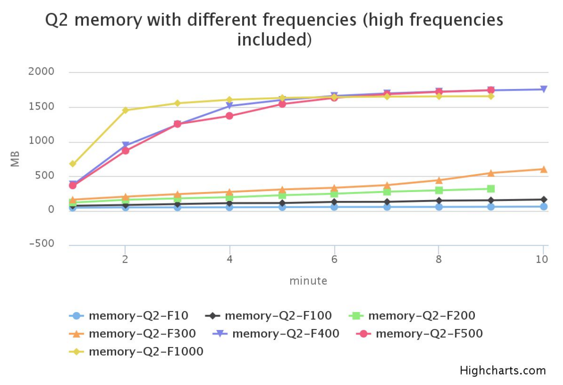


Figure 19: Query 2 latency and memory usage with varying frequencies

As the queries have a step of 5 seconds and new triples are added to the stream every second, the optimal query running time would be 2.5 seconds. With frequency of 10 the latency stays the same in the course of the test. Both queries overall average gets close to that: 2.595s for Q1 and 2.702s for Q2. As the frequency gets higher so does the latency. When frequency is set to 200 the averages for Q1 and Q2 are 3.673s and 5.585s accordingly. At frequency of 300 Q2 starts to rise exceedingly at the 7 minute mark having latency of 7.510s on the first minute and 19.424s at the end of the test. With frequencies of 400 and 500 Q2 stopped registering latency after 3. Minute. Q1 Managed to stay stable at frequency of 1000 with an average latency of 6.943s, but with a frequency of 1250 did not get past the first minute.

With higher frequencies we can see memory flatlining above 1500MB mark and out of memory errors appeared on the application logs, which explains why latencies stopped being registered. When we look at the memory graphs closer, we can see, that the higher the frequency, the more rapidly the memory starts to fill. The main reason for this is that as the latency rises higher than 5s, which is the step length for our queries, more and more data has to be kept in memory. That is because data is discarded only after all queries, in whose windows the data belongs to, have run. But memory usage is rising even for slower frequencies. For example when frequency equals 10, during the 10 minutes, used memory rises 49.55MB - 39.19MB = 10.36MB. With frequency of 200 the rise is 153MB - 86.44MB = 66.56MB.

With regards to Register Stream API requirements: with frequency of 1, 96 triples are streamed in a second - that is 8 294 400 triples a day, which covers our estimation of 2 million triples a day.

5.3.2 Varying window sizes

Figures 20 and 21 show the queries latency and memory consumptions with different window sizes. The frequency and number of registered queries for these tests were 1.

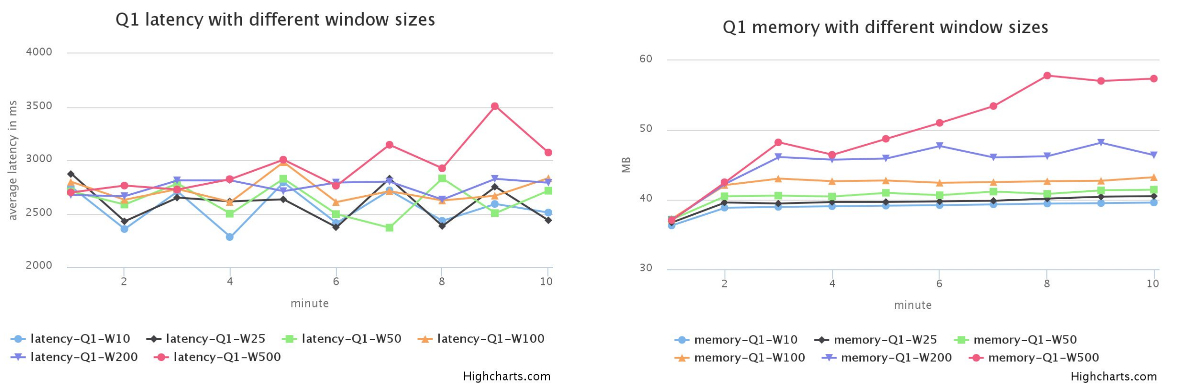


Figure 20: Query 1 latency and memory usage with varying window sizes

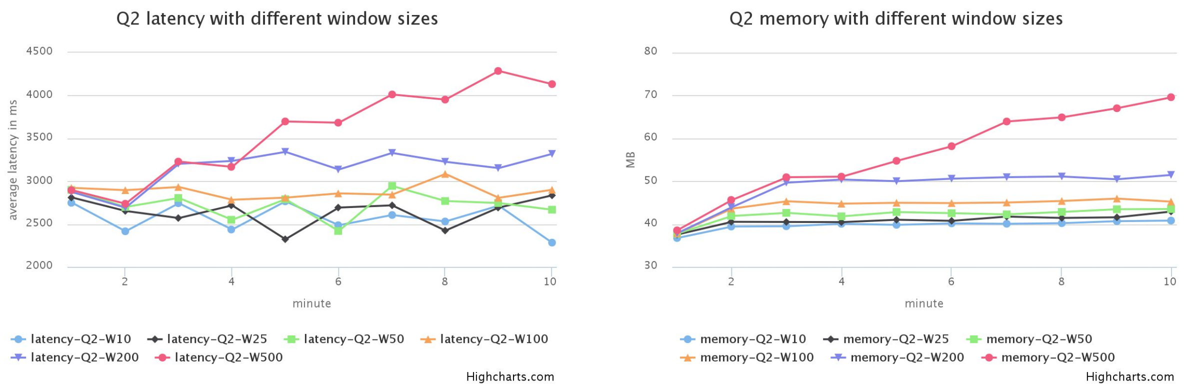


Figure 21: Query 2 latency and memory usage with varying window sizes

The highest window size we measured was 500 seconds - that is 8 minutes and 20 seconds. As the tests ran for 10 minutes, it would have been impractical to set it much higher because the test will end sooner than the window becomes ‘full’.

A noticeable rise in latency can be seen with a window size of 500 seconds for Q1 and 200 seconds for Q2, but it is marginal compared to the frequency tests. This is understandable as with window size 500 seconds and a stream rate of 96 triples per second, the number of triples in memory after 500 seconds is 48 000. This is equivalent for window size of 5 seconds and a frequency of 100. For Q1 the average latency in the first scenario was 2.954s and in the second

2.979s, which are both smaller than the query step of 5 seconds. To strain the system more with larger window sizes in future tests, one could run the tests longer or setting the frequency higher.

In this benchmark triple timestamps are set, when they enter the application. This can also be seen from the graphs, which keep ascending as long as the window size, and then slow down. To imitate large window sizes like 1 year, as might be required for Inforegister NOW! application, one could add a large number of triples to the stream with older timestamps.

5.3.3 Varying registered queries

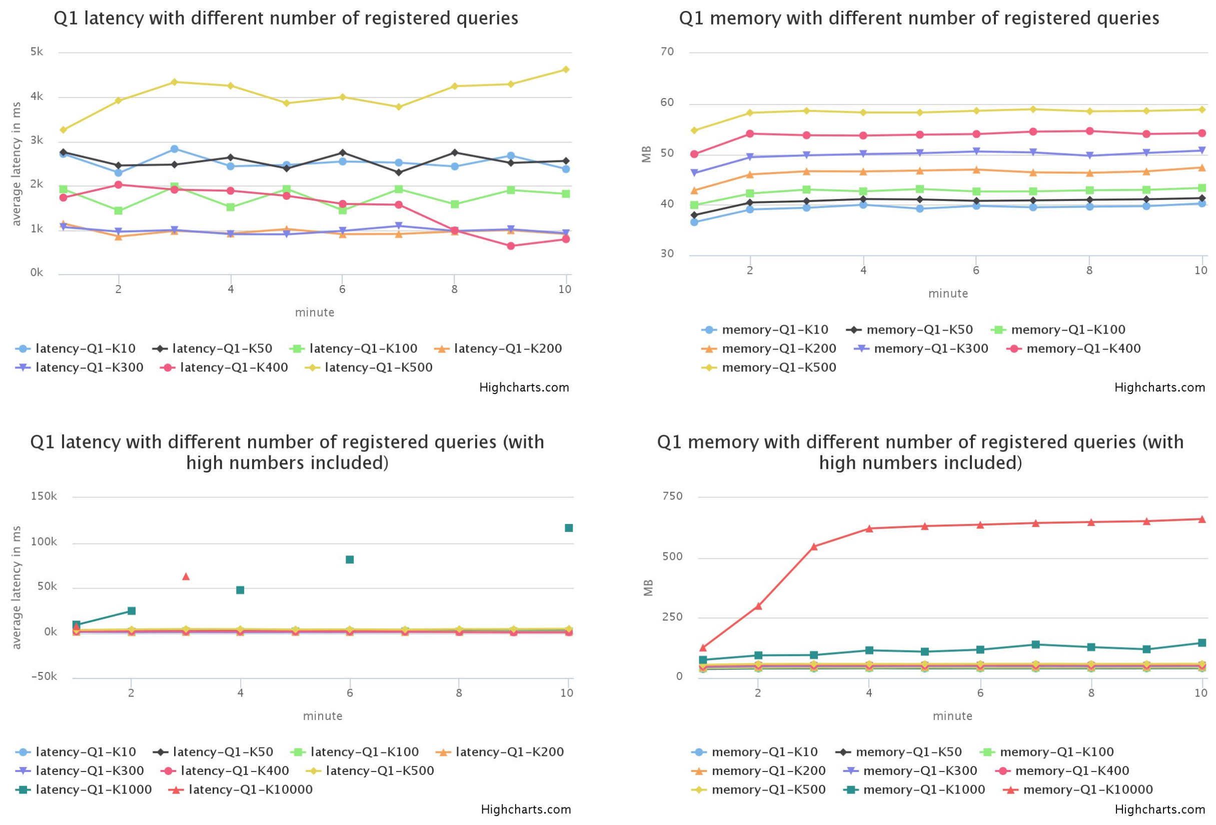


Figure 22: Query 1 latency and memory usage with different number of registered queries.

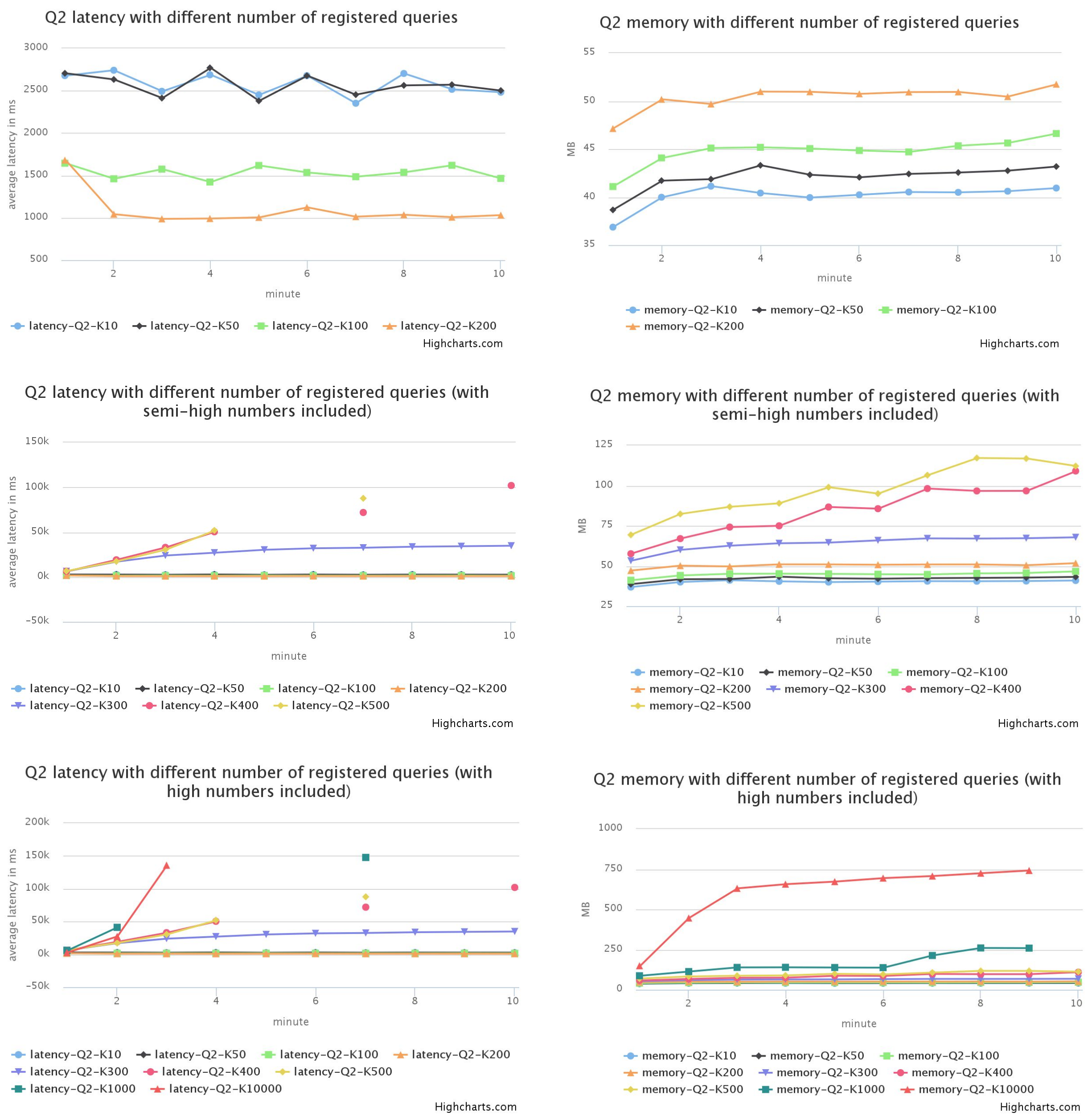


Figure 23: Query 2 latency and memory usage with different number of registered queries.

The latency started climbing more rapidly with 500 queries for Q1 and 300 queries for Q2. This time it is not caused by large number of triples in memory, but because the queries had to wait on each other. When the number of concurrent queries were 10 000 the application started throwing ConcurrentModificationExceptions from the C-SPARQL engine implying that different queries tried to modify the same object at the same time. 10 000 queries is still significantly lower than are expected of Register Stream API. Test results suggest that a safe number of queries for a single stream like Q1 is 400 and for a two stream query like Q2 200.

One way to alleviate this is by having multiple users observe the same query instead of registering identical queries more than once. Another probably more effective way to improve this is to have a clustered system with multiple engines that balance the load of a big number of users. There could be a front application which chooses the engine to use based on the number of queries already registered on them. On the other end RabbitMQ could duplicate the streams to different engines.

5.3.4 Threats to validity

Test results of concurrent queries exhibit some unusual behaviour. There is a range for the number of parallel queries where the average latency is lower than with smaller number. Both queries have latency around 2.5 seconds, but with 100 and 200 queries it is below 2 seconds. This is lower than the optimal 2.5 seconds mentioned earlier. After some inspection of the code, it seems more likely that this anomaly is caused by the benchmark application, rather than the fault of the C-SPARQL engine. Largest number of parallel queries for the tests run by CityBench authors was 20 while on our tests this oddity started appearing after 100 concurrent queries.[26]

Some of this memory surge mentioned discussed in Subsection 5.3.1 with frequencies 10 and 200 could also be caused by the CityBench application itself and not the C-SPARQL engine.

6. Conclusion

In this paper we introduced RDF and SPARQL technologies for linked data and two widely used architecture options for web applications in REST and message-oriented middleware. We gave a brief overview of C-SPARQL - an extension of SPARQL for querying over RDF streams and RabbitMQ - message-oriented middleware that uses primarily AMQP protocol for message mediation. We created two proofs of concepts for RDF stream filtering solutions and proposed a third one, suitable for an enterprise-level application like Inforegister NOW!. It takes advantage of both REST and MOM capabilities and is flexible by offering several methods for registering queries and accessing the results. We also run experiments on C-SPARQL to measure its suitability for Inforegister NOW! Stream API. Even though frequency test results were adequate for the Stream API-s requirements, the tests were to light to strain the system for varying window sizes and both the engine and the benchmark had their shortcomings with regards to having multiple parallel queries run at the same time.

In the future a lot more experiments could be made. To stress the system properly with window size, longer tests could be made with higher triple frequencies. Likewise tests with high frequencies with several streams and queries running at the same time. For example, which would have greater latency: one query and stream with frequency of 1000 or ten queries and streams each with frequency of 100 (assuming here that each streams triple rate would be the same). Tests with multiple observers could also be made although the efficiency of the observer depends on its implementation. For example through the RESTful API of C-SPARQL one could add several observers all pointing to different URL-s where the application would have to send the results, but if there was an observer who would send all the query results to RabbitMQ, then one observer would be enough and the load with different number of users would be the MOM's responsibility.

Other RSP engines that were mentioned in Related Work Section, could also be tested. Both YABench and CityBench have been used to test more than one engine and are made to be modular enough to have engines plugged into them relatively easily. [15], [26]

If enough experiments have been made to meet the requirements of the application, then the architecture design could be updated according to the test results and a prototype could be made.

References

[1] T. Berners-Lee, J. Hendler, O. Lassila, and Others, “The semantic web,” *Sci. Am.*, vol. 284, no. 5, pp. 28–37, 2001.

[2] “Inforegister developers.” [Online]. Available: https://developers.ir.ee/stream-api/. [Accessed: 12-May-2016].

[3] F. Manola, E. Miller, B. McBride, and Others, “RDF primer,” *W3C recommendation*, vol. 10, no. 1–107, p. 6, 2004.

[4] “SPARQL 1.1 Query Language.” [Online]. Available: https://www.w3.org/TR/2013/REC-sparql11-query-20130321/. [Accessed: 12-May-2016].

[5] M. Masse, *REST API design rulebook*. “ O’Reilly Media, Inc.,” 2011.

[6] “Message-Oriented Middleware (MOM) (Sun Java System Message Queue 4.3 Technical Overview).” [Online]. Available: http://docs.oracle.com/cd/E19340-01/820-6424/aeraq/index.html. [Accessed: 12-May-2016].

[7] D. Le-Phuoc, M. Dao-Tran, J. X. Parreira, and M. Hauswirth, “A Native and Adaptive Approach for Unified Processing of Linked Streams and Linked Data,” in *The Semantic Web – ISWC 2011*, Springer Berlin Heidelberg, 2011, pp. 370–388.

[8] D. Anicic, S. Rudolph, P. Fodor, and N. Stojanovic, “Stream reasoning and complex event processing in ETALIS,” *Semantic Web*, vol. 3, no. 4, pp. 397–407, 2012.

[9] J.-P. Calbimonte, O. Corcho, and A. J. G. Gray, “Enabling Ontology-Based Access to Streaming Data Sources,” in *The Semantic Web – ISWC 2010*, Springer Berlin Heidelberg, 2010, pp. 96–111.

[10] S. Komazec, D. Cerri, and D. Fensel, “Sparkwave: continuous schema-enhanced pattern matching over RDF data streams,” *of the 6th ACM International Conference …*, 2012.

[11] M. Rinne, E. Nuutila, and S. Törmä, “Instans: High-performance event processing with standard rdf and sparql,” in *11th International Semantic Web Conference ISWC 2012*, 2012, p. 101.

[12] Y. Zhang, P. M. Duc, O. Corcho, and J.-P. Calbimonte, “SRBench: A Streaming RDF/SPARQL Benchmark,” in *The Semantic Web – ISWC 2012*, Springer Berlin Heidelberg, 2012, pp. 641–657.

[13] D. Le-Phuoc, M. Dao-Tran, M.-D. Pham, P. Boncz, T. Eiter, and M. Fink, “Linked Stream Data Processing Engines: Facts and Figures,” in *The Semantic Web – ISWC 2012*, Springer Berlin Heidelberg, 2012, pp. 300–312.

[14] D. Dell’Aglio, J.-P. Calbimonte, M. Balduini, O. Corcho, and E. Della Valle, “On Correctness in RDF Stream Processor Benchmarking,” in *The Semantic Web – ISWC 2013*, Springer Berlin Heidelberg, 2013, pp. 326–342.

[15] M. Kolchin and P. Wetz, “Demo: YABench-Yet Another RDF Stream Processing Benchmark,” in *RSP Workshop*, 2015.

[16] M. Balduini, E. di Nitto, M. Miglierina, V. Munteanu, G. Casale, J. F. Pérez, and W. Wang, “MODAClouds D6. 3.1-Monitoring platform-initial release.” 2013.

[17] H. Khrouf and X. Ren, “WAVES: Deliverable 2.3 v2 Experiment Infrastructure.” [Online]. Available: http://www.waves-rsp.org/deliverables/Waves-D2.3-expInfrastructure-v2.pdf.

[18] H. N. M. Quoc and D. Le Phuoc, “An Elastic and Scalable Spatiotemporal Query Processing for Linked Sensor Data,” in *Proceedings of the 11th International Conference on Semantic Systems*, 2015, pp. 17–24.

[19] A. Gnoli, *C - SPARQL: A Continuous Query Language for Resource Description Framework Data Streams*. LAP Lambert Academic Publishing, 2010.

[20] D. F. Barbieri, D. Braga, S. Ceri, and M. Grossniklaus, “An Execution Environment for C-SPARQL Queries,” in *Proceedings of the 13th International Conference on Extending Database Technology*,

2010, pp. 441–452.

[21] “RabbitMQ - AMQP 0-9-1 Model Explained.” [Online]. Available: https://www.rabbitmq.com/tutorials/amqp-concepts.html. [Accessed: 12-May-2016].

[22] pubsubhubbub, “pubsubhubbub/PubSubHubbub.” [Online]. Available: https://github.com/pubsubhubbub/PubSubHubbub/wiki. [Accessed: 12-May-2016].

[23] “RabbitHub.pdf.” [Online]. Available: http://commondatastorage.googleapis.com/opensourceprojects/RabbitMQ/RabbitHub.pdf.

[24] T. Kalling and Others, “Järjepideva veebimonitoorimise süsteemi arhitektuur Eesti domeenis,” Tartu Ülikool, 2015.

[25] J. L. Fernandes, I. C. Lopes, J. J. P. C. Rodrigues, and S. Ullah, “Performance evaluation of RESTful web services and AMQP protocol,” in *2013 Fifth International Conference on Ubiquitous and Future Networks (ICUFN)*, 2013, pp. 810–815.

[26] M. I. Ali, F. Gao, and A. Mileo, “CityBench: A Configurable Benchmark to Evaluate RSP Engines Using Smart City Datasets,” in *The Semantic Web - ISWC 2015*, Springer International Publishing, 2015, pp. 374–389.

Appendix 1. Repository of Proof of Concepts.

The source code repository for Proof of Concepts is located at https://github.com/a71993/csparqlpush.