YABench: A Comprehensive Framework for RDF Stream Processor Correctness and Performance Assessment

Maxim Kolchin[1], Peter Wetz[2(✉)], Elmar Kiesling[2], and A Min Tjoa[2]

[1] ITMO University, Saint Petersburg, Russia
kolchinmax@niuitmo.ru
[2] TU Wien, Vienna, Austria
{peter.wetz,elmar.kiesling,a.tjoa}@tuwien.ac.at

Abstract. RDF stream processing (RSP) has become a vibrant area of research in the semantic web community. Recent advances have resulted in the development of several RSP engines that leverage semantics to facilitate reasoning over flows of incoming data. These engines vary greatly in terms of implemented query syntax, their evaluation and operational semantics, and in various performance dimensions. Existing benchmarks tackle particular aspects such as functional coverage, result correctness, or performance. None of them, however, assess RSP engine behavior comprehensively with respect to all these dimensions. In this paper, we introduce YABench, a novel benchmarking framework for RSP engines. YABench extends the concept of correctness checking and provides a flexible and comprehensive tool set to analyze and evaluate RSP engine behavior. It is highly configurable and provides quantifiable and reproducible results on correctness and performance characteristics. To validate our approach, we replicate results of the existing CSRBench benchmark with YABench. We then assess two well-established RSP engines, CQELS and C-SPARQL, through more comprehensive experiments. In particular, we measure precision, recall, performance, and scalability characteristics while varying throughput and query complexity. Finally, we discuss implications on the development of future stream processing engines and benchmarks.

1 Introduction

Major developments such as the Internet of Things, Smart Cities, and Smart Devices increasingly shift data processing challenges from a static towards a continuous paradigm. In many domains, it is crucial to make sense of frequently changing data flows in order to draw timely conclusions about the state of a system. This necessitates means for the efficient processing of and reasoning over dynamic data while accounting for its temporal dimension. Application examples include decision support in a smart city context, environmental monitoring, public transport management, and pervasive healthcare systems.

M. Kolchin and P. Wetz—These authors contributed equally to this work.

© Springer International Publishing Switzerland 2016
A. Bozzon et al. (Eds.): ICWE 2016, LNCS 9671, pp. 280–298, 2016.
DOI: 10.1007/978-3-319-38791-8_16

The notion of continuous queries, which was introduced by Terry et al. [14] in 1992, is central to dynamic data processing. These queries are issued once and executed continuously to provide up-to-date results as new data arrives [7]. Data stream management systems (DSMSs) and their respective query languages, which make use of continuous queries, have been the topic of intense research since 2002 [2]. More recently, dynamic data streams have attracted considerable interest within the semantic web community. This led to the development of various approaches to enable SPARQL-like data access on flows of incoming data under the labels *stream processing* and *stream reasoning*. The resulting RDF stream processing engines, including C-SPARQL [3], CQELS [10], and $\text{SPARQL}_{\text{Stream}}$ [6], combine stream data processing with semantic web technologies. By leveraging the explicit semantics of RDF streams, these engines facilitate reasoning over dynamic data. They also allow to integrate and fuse data streams through federated queries and provide a platform for innovative applications in data stream analytics.

Implementations of these engines result in considerable differences in both performance characteristics [8,11,15] and crucial differences in operational semantics. A common benchmarking framework would help to assess differences and limitations of these existing implementations, but also provide a basis for steering future research directions and standardization efforts. The need for such standardization is highlighted by the formation of an *RDF Stream Processing* community group backed by W3C[1]. Design decisions in this context should be informed by comprehensive benchmarks, which this work aims to provide.

Previous studies analyzed isolated aspects such as functional coverage [15], performance [11], or correctness [8] through specialized benchmarks. This work focuses on window-based stream processing engines and compares them along all these dimensions. To this end, we developed YABench (Yet Another RDF Stream Processing Benchmark), an integrated framework to assess both correctness and performance of RSP engines. YABench provides means for the definition of test scenarios, generates reproducible test data streams, performs evaluation runs, and provides analyses of the results. It provides full reproducibility and emphasizes visual presentation of results to foster an understanding of engines' individual characteristics, including correctness under varying input loads, window sizes, and window frequencies.

The remainder of the paper is organized as follows (contributions highlighted):

- Sect. 2 provides an **overview** of related work and differentiates YABench from other RSP benchmarks and work in related domains;
- Sect. 3 introduces our **stream generator** (Sect. 3.2), discusses currently **supported engines** (Sect. 3.3), and outlines the design of the **oracle** component (Sect. 3.4);
- next, we **validate YABench** against CSRBench (Sect. 4), and
- discuss results of our **experiments** (Sect. 5) that analyze engines' correctness under varying input parameters.

[1] http://www.w3.org/community/rsp/, Accessed Jan. 12th, 2016.

– We conclude with a **discussion** on implications and an outlook on future work in Sects. 6 and 7, respectively.

2 Related Work

In the traditional data streaming domain, the Linear Road (LR) benchmark [1] is widely used for evaluating DSMSs. It is based on a configurable toll system simulation and consists of a historical data generator, a traffic simulator, a data driver, and a validator. LR provides a comprehensive framework for experiments and served as an inspiration for YABench, where our goal is to provide a similarly comprehensive benchmark for RDF stream processing. Like LR, YABench is capable of testing functional aspects such as window-based queries, joins, filters, and aggregations. In addition, YABench covers RSP-specific aspects not covered by LR, such as the influence of query complexity and varying throughput on precision, recall, and delay per window.

In the semantic web domain, there are several well-established benchmarks, including LUBM [9], FedBench [13], BSBM [4], and DBPSP [12]. These benchmarks, however, operate on static knowledge bases and focus on characteristics such as query execution and load times. They do not address aspects that arise in a streaming context, such as correctness under high load. RSP engines' inconsistent interpretation of streaming operators' semantics poses additional challenges and precludes the reuse of existing (non-streaming) benchmarks.

The RSP research community has also developed a number of specialized benchmarks. LSBench [11] first allowed comparisons between Linked Stream Data processing engines. Using a social network scenario, the benchmark uncovered conceptual and technical differences between CQELS, C-SPARQL, and JTALIS. Furthermore, it highlighted performance differences between these engines and included limited functionality and correctness tests. Because LSBench does not include means to determine the correct output, however, it does not provide absolute correctness figures to RSP engine developers. The benchmark is also not customizable for engines' varying execution strategies. YABench overcomes these limitations by introducing a configurable oracle that allows to emulate the behavior of different engines. This is an essential requirement due to the fact that currently available engines do not agree on common operational semantics. Hence, the oracle represents a means to create reproducible results based on configurable operational semantics allowing to compare results from different engines along different dimensions such as performance and correctness.

SRBench (Streaming RDF/SPARQL Benchmark) [15] defines a set of queries that cover RSP-specific aspects, such as ontology-based reasoning or the application of static background knowledge to streaming data. The authors conduct a functional evaluation of C-SPARQL, CQELS, and $\text{SPARQL}_{\text{Stream}}$ and conclude that the capabilities of these engines are still fairly limited. Due to the focus on functional aspects, SRBench does not recognize differences in the operational semantics of the benchmarked systems. To validate the query results, the authors propose correctness metrics such as precision and recall. YABench implements

these metrics on a per-window basis and thereby makes it possible to quantify engines' retrieval performance on the most granular level.

CSRBench (Correctness checking Benchmark for Streaming RDF/SPARQL) [8] focuses on the correctness of stream query results. To this end, CSRBench evaluates RSP engines' compliance to their respective operational semantics using an oracle that determines the validity (i.e., correct or incorrect) of the query results. It thereby complements functional (SRBench) and performance (LSBench) evaluations. The authors find that none of the tested engines passes all tests and provide a detailed account on why certain engines fail at specific queries. CSRBench takes first steps towards validating RSP engines, but lacks comprehensive correctness evaluations over time. YABench extends the idea of oracle-based validation using more comprehensive correctness metrics (i.e., precision and recall) for each window. Moreover, we relate these correctness metrics directly to performance metrics such as delay in query result delivery or memory consumption and CPU utilization. Thereby, YABench provides insights into throughput and scalability and provides a comprehensive toolset to investigate RSP engine characteristics, including both performance and correctness. In addition, our modular architecture also allows researchers to easily exchange the RSP engines, stream generators, and continuous queries used in the benchmark. To the best of our knowledge, YABench is the first RSP benchmarking framework to provide such functionality.

3 YABench Framework

We define the following requirements for comprehensive RSP benchmarking:

- R1 *Scalable and configurable input*: Input data should be scalable and configurable to allow for the flexible definition of benchmark scenarios. This allows researchers to conduct experiments under varying conditions, such as high/low load and varying window sizes.
- R2 *Comprehensive correctness checking*: It should be possible to check the correctness of results, despite engines' varying operational semantics. Moreover we aim at measuring *real throughput*, i.e., how does input load affect correctness of results.
- R3 *Flexible queries*: Queries should be parameterizable, i.e., it should be possible to create test configurations using the same queries, but varying query parameters.
- R4 *Reproducibility*: Experiments need to be reproducible to ensure independent validation of results is possible at a later point in time.

The YABench framework implements these requirements through four elements: (i) a stream generator that create test data streams; (ii) an oracle that tests the correctness of results; (iii) supported engines to be benchmarked; and (iv) a reporting tool that visualizes the results.

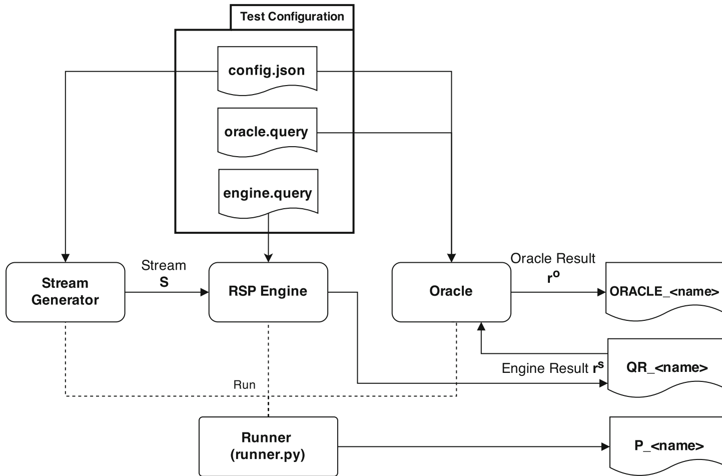


Fig. 1. Architecture of YABench framework

3.1 Architecture

YABench is designed around a modular architecture (Fig. 1) that decouples test configuration from execution. It allows to define tests in a declarative manner and can run complete benchmarking workflows with a single command.

The framework consists of four separately executable modules, i.e., the *Stream Generator*, *RSP engine*, the *Oracle* and the *Runner* which controls the overall execution flow of a test. The test configuration consists of a configuration file (*config.json*) and two query templates, *engine.query* for the engine and *oracle.query* for the oracle. The configuration file defines a set of tests that use the same query templates, but with varying parameters such as window size and slide (*R3*).

Each benchmark yields oracle results ($ORACLE_ < name >$) and performance measurements ($P_ < name >$). These results can be visualized by means of a provided reporting web application. More details about the architecture and the test configuration can be found on the wiki of the project's GitHub repository[2].

3.2 Stream Generator

The *Stream Generator* implements *R1* and is used to create input data that is subsequently fed to the respective engines. It turned out to be more practical to separate the steps of creating data, feeding it to the engines, and creating measurements. The stream generator emulates an environmental monitoring scenario and draws on the LinkedSensorDataset[3] which is also used for SRBench [15] and CSRBench [8]. The data set consists of weather observations from hurricanes in the USA. We selected this simple data model for two reasons: (i) it makes our work

[2] http://github.com/YABench/yabench, Accessed Jan. 12th, 2016.
[3] http://wiki.knoesis.org/index.php/SSW_Datasets, Accessed Jan. 12th, 2016.

comparable to previous work, particularly to CSRBench (see Sect. 4); and (ii) having such a simple and generic model allows for scenario parameterization, e.g., by changing the *number of simulated weather stations* to vary *load on an engine*.

To simulate more complex data flows, YABench can easily be extended with additional stream generators by extending the class `AbstractStreamGenerator` from the *yabench-generator* module[4]. Note however, that more complex data streams increasingly make it difficult to isolate effects (i.e., which parameter influences which measurable performance indicator).

Figure 2 illustrates the structure of the data model. A central element of the data model is `weather:TemperatureObservation`, which represents a single observation. This observation is connected to actual measure data via `om-owl: result`. The `om-owl:observedProperty` indicates which environmental condition was sensed by the `ssw:system`. The system represents a sensor which is creating measurements. It is connected to the observation via the `om-owl:procedure` relation.[5]

Based on an input function $gen(s, i, d, r, n)$, the process generates an RDF stream $\mathbb{S}$ where s denotes the number of simulated systems, i denotes the time interval between two measurements of a single station, d denotes the duration of the generated output stream, r denotes a seed for randomization to vary the timestamps of initial measurements of each system, and n defines the generator which should be used. Input load for experiments can be varied with the s and i parameters. The combination of parameters s, i, and r ensures reproducibility, because the stream generator guarantees that the exact same stream is generated every time for a given parameter set, hence, this satisfies *R4*.

3.3 Engines

YABench can currently benchmark two engines, i.e., C-SPARQL 0.9.5[6], and CQELS 1.0.0[7]. After a stream has been generated, the YABench engine component calls a function $stream(d, q, s)$, where d denotes the destination of output files, q defines the continuous query which will be registered at the engine, and s defines the input stream, which was previously generated. At this stage output files will be created for performance measurements and query results.

YABench essentially wraps each engine to allow stream data feeding under controlled conditions. The input RDF stream $\mathbb{S}$ consists of a sequence of timestamped triples in non-decreasing time order in the form of $\mathbb{S} = ((\langle s, p, o), t_0\rangle), (\langle s, p, o), t_1\rangle), ...)$. The wrapper iterates over the input stream $\mathbb{S}$ and feeds sets of RDF statements with same timestamps to the engine, hence,

[4] https://github.com/YABench/yabench/tree/master/yabench-generator, Accessed Jan. 12th, 2016.

[5] The authors are aware of the fact that at the time of writing, two properties of the used vocabulary for the data model have undergone quasi-standardization as part of the SSN ontology and were changed in the process (`result` changed to `observationResult` and `procedure` changed to `observedBy`).

[6] http://github.com/streamreasoning/CSPARQL-engine, Accessed Jan. 12th, 2016.

[7] https://code.google.com/p/cqels/, Accessed Jan. 12th, 2016.

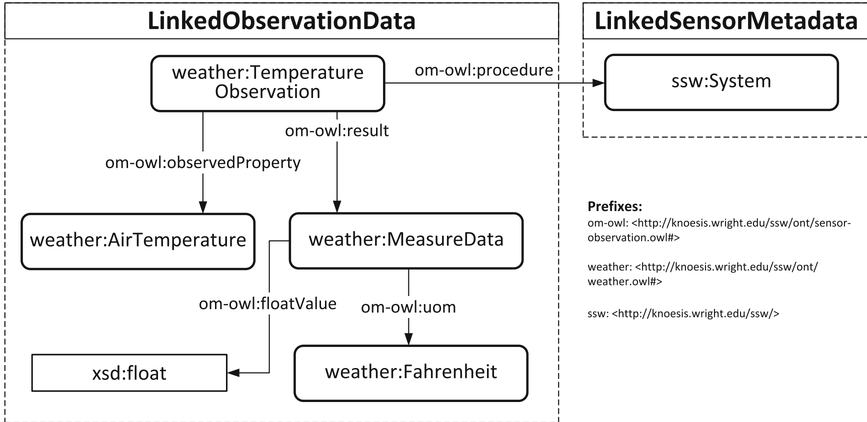


Fig. 2. Data model of generated streams based on LinkedSensorData.

$F_{t_0} = \{(\langle s, p, o)\rangle \mid (\langle s, p, o), t_0\rangle \in \mathbb{S}\ \}$ while respecting time intervals $i = t_1 - t_0$ between feeds F_{t_0} and F_{t_1}.

While feeding the engines with the graphs, we continuously take measurements, i.e., absolute and relative memory consumption, cpu usage, and the number of threads spawned. Because YABench is implemented in Java, additional Java-based RSP engines can be easily integrated for benchmarking. To this end, it is sufficient to extend the classes `AbstractEngine`, `AbstractQuery`, `ContinuousListener`, and `AbstractEngineLauncher`, all of which are available in the *yabench-commons* module[8].

3.4 Oracle

To check the correctness of query results and thereby satisfy *R2*, we implemented an oracle. The implementation is inspired by the oracle used in CSRBench [8], but built on top of Jena ARQ[9] and extended with more granular metrics, which are computed for each window: precision and recall, delay of query results, number of triples in the window, and number of tuples in the query results.

The semantics used by the oracle can be configured, i.e., the specification of the oracle can be changed in accordance to the benchmarked engine. Report policy parameters [5] are provided to emulate either CQELS (*OnContentChange*) or C-SPARQL (*OnWindowClose*). Hence, we are able to provide correct results that account for the respective report policy.

The oracle checks the results (recorded in $QR_ < name >$ file) of a continuous query q by using the same input stream $\mathbb{S}$ and an equivalent, but static, SPARQL query q' (*oracle.query*). It takes into account the report policy which the given

[8] https://github.com/YABench/yabench/tree/master/yabench-commons, Accessed Jan. 12th, 2016.

[9] https://jena.apache.org/documentation/query/, Accessed Jan. 12th, 2016.

engine applies as well as window size α and slide β parameters of the continuous query q. The following report policies are provided by the oracle:

- *Content change*: reporting when the content of the active window changed (used by CQELS).
- *Window close*: reporting when the active window closes (used by C-SPARQL).

The oracle uses the following procedure to check correctness:

1. Determine the *scope* $[t_s, t_e)$ of the next window that will report based on the given window size α, window slide β, and report policy.
2. Use the *scope* $[t_s, t_e)$ to select window *content* from the input stream $\mathbb{S}$ where the relevant triples are $F_{t_s,t_e} = \{(\langle s,p,o\rangle) \mid (\langle s,p,o\rangle, t\rangle \in \mathbb{S}, t_s \leq t < t_e\}$.
3. Compute the expected result by executing the SPARQL query q' on F_{t_s,t_e} on the query engine.
4. Compare the result of this query with the next result of continuous query q and compute precision/recall metrics.
5. Compute remaining metrics, i.e., delay, window size, and result size.

Delay. The delay d of query results is defined as the difference between the end timestamp of the oracle window $t_{e_o}^{\mathbb{W}_i}$ and the actual timestamp $t_{e_s}^{\mathbb{W}_i}$ of the engine's output for this window. The timestamp when the engine output the result is recorded in file $QR_ < name >$.

Gracious Mode. In addition to the algorithm described above, the oracle implements a new *gracious* mode to reveal issues associated with wrong window content. In the default *non-gracious* mode, the oracle strictly uses the defined window size and range from the registered continuous queries to calculate the content of windows which, in turn, is used to measure precision and recall. However, we found that low precision and recall are often caused by imprecise event processing near windows borders. To determine the magnitude of this effect and isolate incorrect query results that are not caused by window border issues, we implemented a *gracious* mode. In this mode, the oracle iteratively shifts window borders to determine for which borders an engine achieves maximum precision and recall. This method allows to reconstruct the actual window borders which were applied by the engines at the time of running the experiments. By looking at the reconstructed windows (see results of experiment four in Sect. 6) researchers can gain visual and quantifiable insights into the extent and cause of incorrect RSP engine behavior.

4 Validation Against CSRBench

By validating YABench against CSRBench, we ensure that it produces equivalent results. In later experiments (see Sect. 5), we then show that our framework

facilitates evaluations that go beyond the scope of CSRBench. The source code and instructions on how to run the validation are published on GitHub[10].
The methodology for the validation is as follows:

– We convert the original data used by CSRBench to N-Triples format, and extend it with timestamps to emulate the input stream load. By doing so, the data stream looks the same as the output of our stream generator.
– For each of the seven CSRBench queries we need to setup tests configurations. These configuration files define parameters such as window size, window slide, and filter parameters, which will be used when registering the queries.
– For each engine and for each of the seven CSRBench queries we then execute the test, i.e., feeding the engines with the input stream and registering queries with the defined parameters.
– After all tests are complete, we compare the results of CSRBench[11] with the results created by YABench.

Table 1. Results of YABench validation against CSRBench results. Cells which include asterisks are referred to in the text.

Query	C-SPARQL		CQELS	
	CSRBench	YABench	CSRBench	YABench
Q1	✓	✓	✓	✓
Q2	✓	✓	✓	✓
Q3	✓	✓*	✓	✓
Q4	✓	✓	×	×
Q5	×	×	✓	✓
Q6	✓	✓*	×	×
Q7	✓	✓*	×	×**

Table 1 summarizes the results of the validation. Results were compared and inspected manually. Checkmarks indicate that YABench produced equivalent results to CSRBench. Columns of CSRBench which contain an $\times$ denote that the respective engine did not produce correct results. In all such cases, YABench also indicated that the engine did not deliver correct results.

Checkmarks denoted with one asterisk (*) indicate that YABench produced largely identical results, but that some results were missing. This occurred in some cases were triples were very close to a window border. In these cases, we found that in C-SPARQL such triples may fall either into the scope of $\mathbb{W}_n$ or $\mathbb{W}_{n+1}$. This can be attributed to timing discrepancies, which we encountered

[10] https://github.com/YABench/csrbench-validation, Accessed Jan. 12th, 2016.
[11] Obtained from https://github.com/dellaglio/csrbench-oracle-engines, Accessed Jan. 12th, 2016.

when running benchmarks multiple times and/or on different systems. However, we verified that all results are present, if not in the correct window, then at the latest in the subsequent window.

Crosses in the YABench columns indicate that results did not match those of CSRBench experiments. This is expected behavior, because the same queries did not pass correctness tests of CSRBench in the original tests either. The cross denoted with two asterisks (**) indicates that the query (*Q7*) did not execute successfully on the CQELS engine.[12]

We can conclude that, besides minor, well-explained inconsistencies, YABench reproduces the results of CSRBench. Beyond the scope of previous benchmarks, however, YABench employs a more comprehensive approach that allows (i) to define experiments including test data, input load parameters, and queries, (ii) to perform experiments that consider the varying operational semantics of the tested engines, and (iii) to conduct in-depth analyses based on new throughput, delay, and correctness metrics. These capabilities are used for the experiments discussed in the following section.

5 Experimental Setup

We use YABench and its oracle to perform experiments with two engines, C-SPARQL and CQELS. In particular, we are interested in how the correctness of results is affected by changes in the input data streams. Whereas previous correctness metrics exclusively focused on checking whether engine results are included in the oracle results, i.e., a yes/no evaluation, we provide more granular metrics. To this end, we use precision and recall calculations in combination with performance metrics. These metrics uncover issues that can be caused by, for instance, shifting of window borders under load, leading to lower precision or recall values. Moreover, we measure an engine's delay in delivering results and the amount of RDF statements inside a window's scope to understand and to explain low retrieval rates. Hence, YABench is the first RSP benchmark to provide a comprehensive picture of an engine's behavior under stress.

Generally, we reuse the queries introduced by CSRBench, however, for each query we are able to parametrize window size α, window slide β, and filter values f thereby satisfying *R3*. For the input streams, we control the number of stations s to simulate low, medium, and high load scenarios. The interval time i between measurements will be 1 s and the duration of each experiment is 30s. The experiments are run in *non-gracious* mode, unless otherwise stated.

Performance measurements, such as memory consumption, are taken at regular intervals, i.e., 500 ms. Because all other metrics observe characteristics of the windows, they are taken and displayed on a per window basis. We replicated each experiment ten times and illustrate the distribution of result metrics obtained for precision, recall, and delay as boxplots.

[12] The system crashed before returning the query results.

We use the CSRBench queries available on the W3C wiki[13] for our experiments, which were executed on an Intel Core i7-3630QM @ 2.4 GHz, Quad Core, 64bit, 12 GB RAM running Windows 7 Professional. In parantheses we denote which queries of CSRBench are reused in each experiment. Complete results are published on GitHub[14] and can be visualized with our web application *YABench-reports*.

5.1 Experiment 1 (*Q1*)

This experiment uses a simple `SELECT` statement combined with a `FILTER` asking for the latest temperature observations above a specified threshold and the sensor which took the measurement. We run the experiment with the following parameters: $\alpha = 5\,s$, $\beta = 5\,s$, $s = 50/1000/10000$ (small/medium/big), $i = 1\,s$.

5.2 Experiment 2 (*Q4*)

The second query makes use of the aggregation function `AVG` combined with a `FILTER` to return the average temperature value over a given window. To answer such aggregate queries, depending on the report policy as well as window content and window size, stream processors typically face high resource costs. Because CQELS does not comply with the semantics of `AVG` as defined by SPARQL 1.1[15], we had to implement a custom `AVG` operator that returns an empty result if there are no matches. We run the experiment with the following parameters: $\alpha = 5\,s$, $\beta = 5\,s$, $s = 1/1000/10000$, $i = 1\,s$.

5.3 Experiment 3 (*Q6*)

This query returns sensors that made two observations (of different timestamps) with a variation between measurements higher than a given threshold. The query uses the `SELECT` keyword to ask for shifts of measured values over time from the same sensor. To execute this query, engines must be capable of joining triples over different timestamps. In order to produce meaningful and comparable results for both engines in this experiment, we slightly changed the number of simulated stations (s) and ran the experiment with the following parameters: $\alpha = 5\,s$, $\beta = 5\,s$, $s = 50/200/500$, $i = 1\,s$.

5.4 Experiment 4 (*Q6*)

This experiment is designed to reveal issues of lower precision and recall values encountered for both engines in particular cases. When running experiment three we observed deteriorating precision and recall due to the following reasons: For CQELS the reason is delayed deletion of window content, for C-SPARQL slight

[13] http://www.w3.org/wiki/CSRBench, Accessed Jan. 12th, 2016.
[14] http://github.com/YABench/yabench-one, Accessed Jan. 12th, 2016.
[15] http://www.w3.org/TR/sparql11-query/#defn_aggAvg, Accessed Jan. 12th, 2016.

window shifts are responsible. This led us to the development of a so called *gracious* mode where the oracle eliminates these issues resulting in both high precision and recall and the possibility to detect new issues unrelated to potential window delays. Hence, this experiment, shows the effects of the *gracious* mode by comparing its results with the *non-gracious* mode as well as discussing and explaining the differences. In *non-gracious* mode the oracle does not account for any issues and expects ideal behavior of the engines.

We ran two similar tests for both engines with the following parameters, one of them in *gracious* mode and the other one in *non-gracious* mode: $\alpha = 5\,s$, $\beta = 5\,s$, $s = 1$, $i = 1\,s$.

6 Discussion

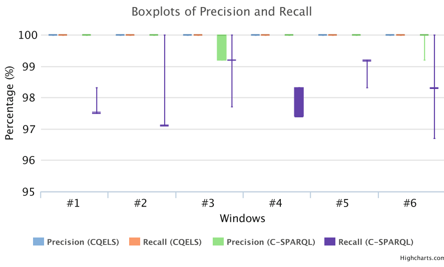
The YABench framework provides a reporting web application named *YABench-reports*. Based on results of the oracle and performance measurements it displays three graphs, i.e., (i) a precision/recall graph that includes indicators for the windows, (ii) a graph showing delay of result delivery, and expected/actual result size, and (iii) a graph providing performance measurements. For the sake of brevity we only discuss the results of experiments one and four in this paper. Experiments two and three are discussed in an online appendix[16].

Figure 3 illustrates the results of the first experiment; Figs. 3a–c show boxplots of precision and recall values for each of the three load scenarios (small: $s = 50$, medium: $s = 1000$, and big: $s = 10000$); Figs. 3d–f show boxplots of the observed delay; and Figs. 4a–c show line charts of an engine's memory consumption during the stream feeding and query evaluation.

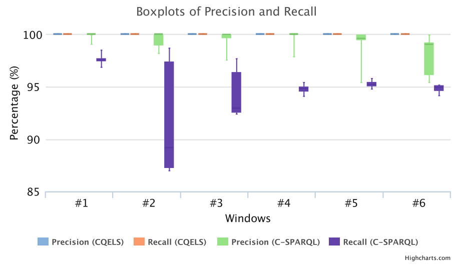
We found that CQELS maintains 100 % precision and accuracy under small load, whereas C-SPARQL achieves slightly lower values (precision is at 100 % except for window three and four, the mean for recall ranges between 97 % and 100 %). Generally, we observe that recall is lower than precision for C-SPARQL. This is due to the shifting of the actual engine windows compared to the ideal expected windows from the oracle due to delays and will be explained in more detail later (see Fig. 5).

We observed similar behavior under medium and high load (see Figs. 3b–c). For this simple query, CQELS still scores perfect precision and recall, whereas we observe deteriorating effects for C-SPARQL. The recall values from window two and three show a particularly higher spread. The mean of all recall measurements is still high. The spread can be explained by the higher delays of the first two windows (see Fig. 3e). Because C-SPARQL delivers results upon the closing of a window, the delay has an effect on precision and recall. This is not the case in CQELS, where delay in result delivery does not necessarily mean that the window content – and consequently the computed results – are incorrect. In fact, for CQELS the opposite is the case, meaning that delayed results still provide correct results. This is also reflected by our oracle.

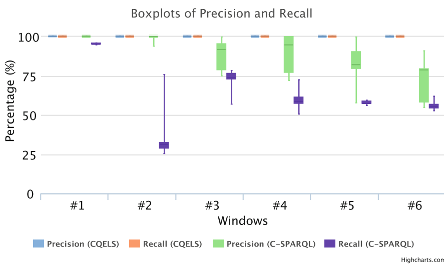
[16] https://github.com/YABench/yabench-one, Accessed Jan. 12th, 2016.



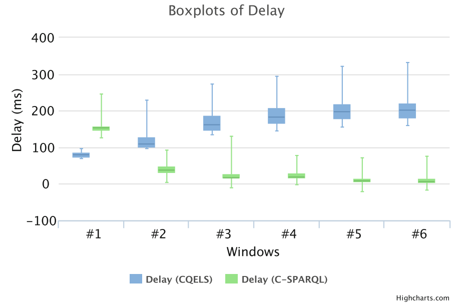
(a) *Q1* precision/recall, $s = 50$



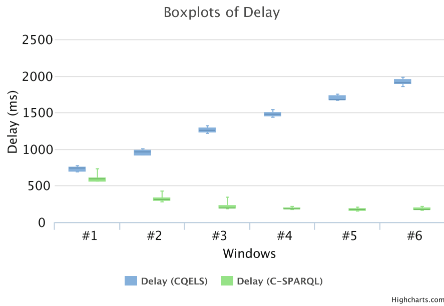
(b) *Q1* precision/recall, $s = 1000$



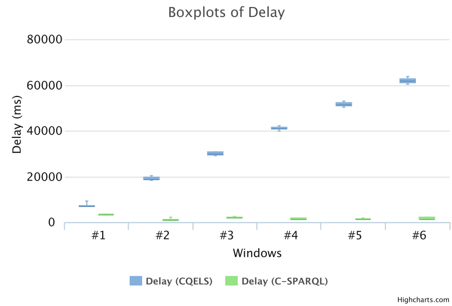
(c) *Q1* precision/recall, $s = 10000$



(d) *Q1* delay, $s = 50$



(e) *Q1* delay, $s = 1000$

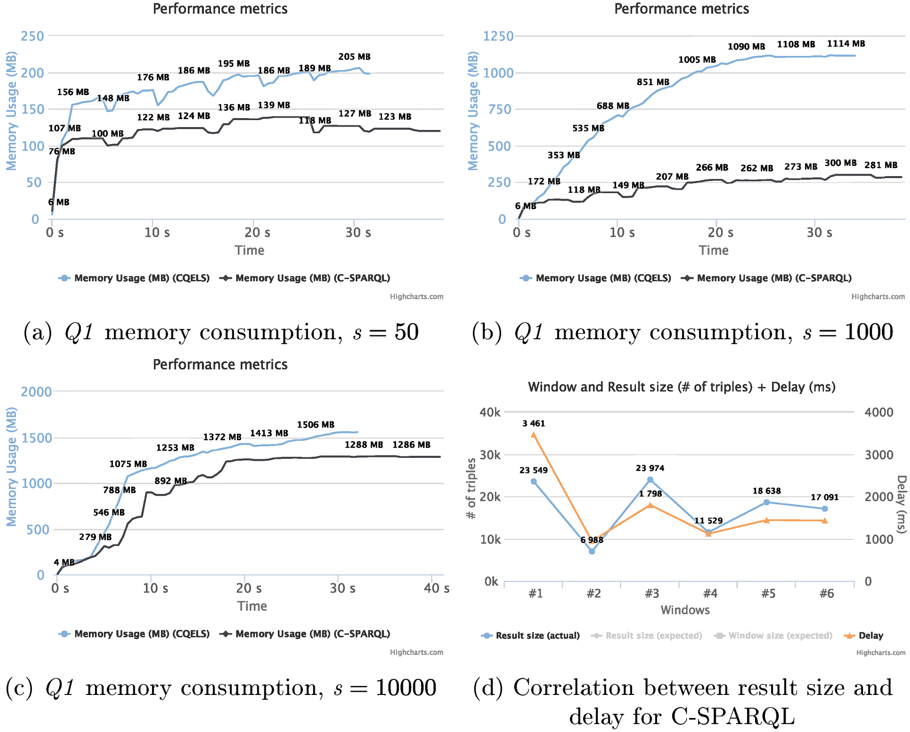


(f) *Q1* delay, $s = 10000$

Fig. 3. Precision/recall results of *Experiment 1* for CQELS and C-SPARQL

Delay of result delivery under low load is depicted in Fig. 3d. For CQELS mean result delivery varies between 81.5 ms and 201.5 ms. The values rise steadily, but even out for the last three windows. Interestingly, delay in C-SPARQL exhibits the opposite characteristics. The first window always yields longer delay (mean = 153.5 ms), whereas the following windows show short delays between 7 ms and 38 ms. Delay can also be negative, when results are delivered earlier than expected.

Figures 3e and 3f show the delay for medium and big load respectively. We observe similar behavior as before, but on a much higher scale. Moreover, CQELS delays do not even out anymore for the last windows, but continue to increase.



(a) *Q1* memory consumption, $s = 50$

(b) *Q1* memory consumption, $s = 1000$

(c) *Q1* memory consumption, $s = 10000$

(d) Correlation between result size and delay for C-SPARQL

Fig. 4. Performance results of *Experiment 1* for CQELS and C-SPARQL

Finally, we found that YABench reveals a correlation between the result size and delay times, as shown in Fig. 4d, which shows both metrics for a single test run. We see that delay times increase when result size increases. This relation is expected, but YABench allows to quantify the influence of large amount of result bindings on an engine's performance.

Figures 4a–c provide details about both engines' performance. C-SPARQL is more memory efficient and exhibits a moderate increase in memory consumption between low (mean = 123 MB) and medium (mean = 250 MB) load. For both engines, the removal of window content is apparent in the charts – particularly in Fig. 4a– in the form of rapid decreases after every five seconds, i.e., the defined window size. Under medium load, memory consumption of CQELS rises to about 1100 MB where it then flattens out. Both engines show similar behavior under high load (see Fig. 4c), where the charts show a steep increase in memory consumption until ten seconds. Beyond that, the graph flattens out again with a maximum of 1506 MB (CQELS) and 1288 MB (C-SPARQL).

As already noted, for C-SPARQL precision always stays higher than recall, if both values deteriorate. The reason why precision and recall decrease, is given by the fact that higher load leads to bigger delays in query result delivery.

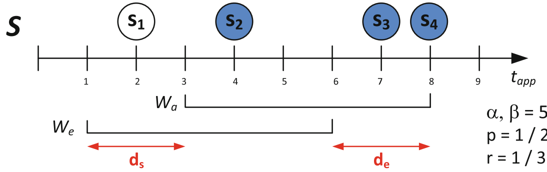


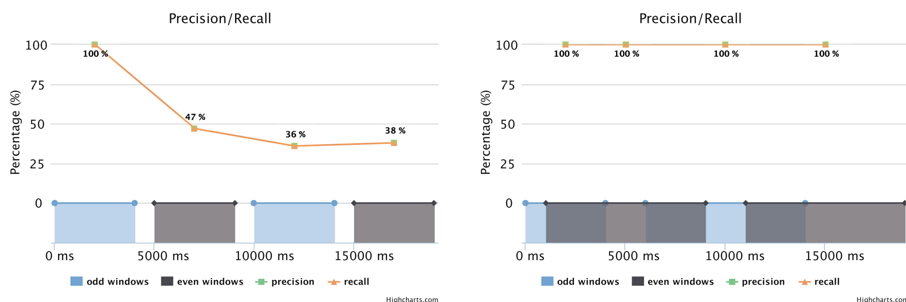
Fig. 5. Lower precision and recall due to delay of actual window $\mathbb{W}_a$

The observed delay supports the conclusion that the actual windows are shifted, therefore, deviating from the ideal windows computed by the oracle as is shown in Fig. 5. Given a query which asks for all statements occurring on stream $\mathbb{S}$, a delay between start and end timestamps of the expected window computed by the oracle $\mathbb{W}_e$ and the actual window $\mathbb{W}_a$ by the engine, can be observed. This is also the reason for lower precision and recall values. $\mathbb{W}_e$ contains only one (s_2) of three relevant statements (blue filling), hence, recall $r = 1/3$. Out of the two selected statements of $\mathbb{W}_e$ ($\{s_1, s_2\}$) only the latter one is relevant, hence, precision $p = 1/2$. In other words, whereas the scope of an ideal window is $[t_s, t_e)$, the scope of shifted windows adds a delay to the start and end timestamps and is denoted as $[t_s + d_s,\ t_e + d_e)$. It is worth to note that d_s and d_e can be different due to timing issues of engines. This explains different declines in precision and recall.

In experiment four, we investigate and explain issues that we experienced while manually testing the engines. Under certain conditions, which are emulated in this experiment, we observed that precision and recall values are decreasing. For CQELS, which employs a *content change* report strategy, these lower values are caused by delayed purging of active window content. By purging, we mean that all elements are deleted from the content of a window. As time in a streaming setting moves on, elements exit the scope of windows, hence, engines are responsible of correctly maintaining the content of windows. In C-SPARQL, which employs a *window close* report strategy, the values can be explained by the shifting of an engine's active window forward on the timeline (see Fig. 5).

To investigate the root causes of these issues we implemented the *gracious* mode. In this mode, the oracle adjusts its window scope to match the scope of an actual window, even though the actual window may contain incorrect elements (see Sect. 3.4). This has two consequences: First, precision and recall values increase, because *gracious* mode reverts the effects of incorrect window content. This allows us to confirm our assumptions on why low precision and recall values were observed. Second, we are able to reconstruct and visualize the window borders that were actually used internally by the engines. Thereby, we unveil differences between expected and actual windows. Expected windows are windows which we would expect from a correctly implemented engine with zero delays.

Figure 6 shows the oracle results of the tests for CQELS. As we can see in Fig. 6a, precision and recall values decrease after the first window for CQELS in *non-gracious* mode. This confirms the issues we experienced when performing



(a) CQELS results in *non-gracious* mode (b) CQELS results in *gracious* mode

Fig. 6. *Experiment 4* results for CQELS in *gracious* and *non-gracious* mode

manual testing. Figure 6b shows results of the same test, but with *gracious mode* enabled. Here we can see that precision and recall values are both at 100 %. YABench achieves this by shifting the window borders of the oracle until reaching maximum precision and recall values. As a result we can see the adapted window borders at the bottom of the charts. The oracle had to shift the window starts to the left in order to reach high precision and recall. This indicates that the engine forgets to delete outdated elements from the content of the active window for the query which is used in this experiment.

Finally, Table 2 presents observations which we made while executing experiments on CQELS. It shows the time the final result was delivered of the tests for small (S), medium (M), and big (B) load scenarios. One would expect the final result to arrive immediately after the last triple was streamed to the engine, which equals the duration of one test, i.e., 30 s in our case. This is the case when we put CQELS under low load as can bee seen in the columns denoted by an *S*. However, under medium and high load, denoted by *M* and *B* columns, we see that delivery delay of the results grows. The reason for that is that CQELS uses the *OnContentChange* report strategy, where queries are evaluated after each streamed triple. Obviously, more complex queries increase the time needed for computation of query results, resulting in a progressive increase in delay. Hence, by showing the arrival of the last result, we can quantify and infer the influence

Table 2. Arrival time (average, minimum, maximum) of final results in seconds for each conducted experiment (E1, E2, E3) under different loads (S = small, M = medium, B = big) with CQELS.

	E1 (SELECT)			**E2** (AVG)			**E3** (SELECT + JOIN)		
	S (50)	**M** (1000)	**B** (10000)	**S** (50)	**M** (1000)	**B** (10000)	**S** (50)	**M** (1000)	**B** (10000)
AVG	30 s	32 s	98 s	31 s	432 s	12966 s	31 s	62 s	269 s
MIN	30 s	32 s	97 s	31 s	426 s	12305 s	31 s	58 s	263 s
MAX	30 s	33 s	100 s	31 s	440 s	13523 s	31 s	65 s	277 s

of different query types on the capability of CQELS to provide timely results. These numbers should not be compared with C-SPARQL where such delays did not appear due to its report strategy, where queries are only evaluated when a window closes resulting in much lower computational effort.

7 Conclusions and Future Work

In this paper, we introduced YABench, a comprehensive RSP benchmarking framework that provides detailed insights into RSP engines' performance and correctness characteristics. These insights are derived from granular metrics, including those that capture engines' capability to produce correct results under load. The framework supports a complete benchmarking workflow from defining tests, generating suitable test data, executing tests, and finally analyzing the results. We have shown that the framework replicates the basic results of an existing benchmark (i.e., CSRBench) and conducted and discussed four more comprehensive experiments, each of which focused on particluar aspects of RSP engines. The resulting visualizations provide insightful information on the characteristics of the tested engines and highlight key differences. In the process of our benchmarks, we also identified and discussed previously unknown issues.

To sum up our findings, YABench reveals that C-SPARQL operates more memory-efficiently than CQELS in all experiments. Both engines perform similarly in terms of delay, but C-SPARQL outperforms CQELS when more complex queries are used and under increasing input load. This can mainly be attributed to the different report strategies implemented by the engines. Concerning precision and recall, CQELS yields better results for simple queries. However, we identified an issue in CQELS which results in decreasing precision and recall measurements. On the other hand, C-SPARQL suffers from window delays, which increase when load on the engine is raised. By introducing a *gracious* mode for running the oracle, we are able to estimate the extent of these effects.

There are several directions for future research. First, we plan to extend functional coverage of the test cases. It would be interesting to evaluate the influence of multiple windows in one query on an engine. To this end, it would be necessary to extend the oracle to support other window operators and combinations of multiple windows. Second, we aim to support benchmarks that involve background knowledge, multiple input streams, and multiple queries. This will broaden our understanding of how well engines can deal with merging high-frequency data streams with large static data sources, which is one of the promising application scenarios for RSP engines.

Finally, we aim to obtain further insights on how engines cope with variations in inter-arrival times of elements.

Acknowledgments. The work done by Peter Wetz was partially funded by TU Wien through the Doctoral College Environmental Informatics. The work done by Maxim Kolchin was partially funded by the Government of the Russian Federation, Grant 074-U01.

References

1. Arasu, A., Cherniack, M., Galvez, E., Maier, D., Maskey, A.S., Ryvkina, E., Stonebraker, M., Tibbetts, R.: Linear road: a stream data management benchmark. In: Proceedings of the 30th International Conference on Very Large Data Bases, VLDB 2004, VLDB Endowment, vol. 30, pp. 480–491 (2004)
2. Babcock, B., Babu, S., Datar, M., Motwani, R., Widom, J.: Models and issues in data stream systems. In: Proceedings of the 21st ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2002, NY, USA, pp. 1–16. ACM, New York (2002)
3. Barbieri, D.F., Braga, D., Ceri, S., Della Valle, E., Grossniklaus, M.: C-SPARQL: a continuous query language for RDF data streams. Int. J. Semant. Comput. **4**(01), 3–25 (2010)
4. Bizer, C., Schultz, A.: The Berlin SPARQL benchmark. Int. J. Semant. Web Inf. Syst. **5**(2), 1–24 (2009)
5. Botan, I., Derakhshan, R., Dindar, N., Haas, L., Miller, R.J., Tatbul, N.: SECRET: a model for analysis of the execution semantics of stream processing systems. Proc. VLDB Endow. **3**(1–2), 232–243 (2010)
6. Calbimonte, J.-P., Corcho, O., Gray, A.J.G.: Enabling ontology-based access to streaming data sources. In: Patel-Schneider, P.F., Pan, Y., Hitzler, P., Mika, P., Zhang, L., Pan, J.Z., Horrocks, I., Glimm, B. (eds.) ISWC 2010, Part I. LNCS, vol. 6496, pp. 96–111. Springer, Heidelberg (2010)
7. Cugola, G., Margara, A.: Processing flows of information: from data stream to complex event processing. ACM Comput. Surv. **44**(3), 15 (2012)
8. Dell'Aglio, D., Calbimonte, J.-P., Balduini, M., Corcho, O., Della Valle, E.: On correctness in RDF stream processor benchmarking. In: Alani, H., et al. (eds.) ISWC 2013, Part II. LNCS, vol. 8219, pp. 326–342. Springer, Heidelberg (2013)
9. Guo, Y., Pan, Z., Heflin, J.: LUBM: a benchmark for OWL knowledge base systems. Web Semant. **3**(2–3), 158–182 (2005)
10. Le-Phuoc, D., Dao-Tran, M., Xavier Parreira, J., Hauswirth, M.: A native and adaptive approach for unified processing of linked streams and linked data. In: Aroyo, L., Welty, C., Alani, H., Taylor, J., Bernstein, A., Kagal, L., Noy, N., Blomqvist, E. (eds.) ISWC 2011, Part I. LNCS, vol. 7031, pp. 370–388. Springer, Heidelberg (2011)
11. Le-Phuoc, D., Dao-Tran, M., Pham, M.-D., Boncz, P., Eiter, T., Fink, M.: Linked stream data processing engines: facts and figures. In: Cudré-Mauroux, P., et al. (eds.) ISWC 2012, Part II. LNCS, vol. 7650, pp. 300–312. Springer, Heidelberg (2012)
12. Morsey, M., Lehmann, J., Auer, S., Ngonga Ngomo, A.-C.: DBpedia SPARQL benchmark – performance assessment with real queries on real data. In: Aroyo, L., Welty, C., Alani, H., Taylor, J., Bernstein, A., Kagal, L., Noy, N., Blomqvist, E. (eds.) ISWC 2011, Part I. LNCS, vol. 7031, pp. 454–469. Springer, Heidelberg (2011)
13. Schmidt, M., Görlitz, O., Haase, P., Ladwig, G., Schwarte, A., Tran, T.: FedBench: a benchmark suite for federated semantic data query processing. In: Aroyo, L., Welty, C., Alani, H., Taylor, J., Bernstein, A., Kagal, L., Noy, N., Blomqvist, E. (eds.) ISWC 2011, Part I. LNCS, vol. 7031, pp. 585–600. Springer, Heidelberg (2011)

14. Terry, D., Goldberg, D., Nichols, D., Oki, B.: Continuous queries over append-only databases. In: Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 1992, NY, USA, pp. 321–330. ACM, New York (1992)
15. Zhang, Y., Duc, P.M., Corcho, O., Calbimonte, J.-P.: SRBench: a streaming RDF/SPARQL benchmark. In: Cudré-Mauroux, P., et al. (eds.) ISWC 2012, Part I. LNCS, vol. 7649, pp. 641–657. Springer, Heidelberg (2012)