A Comparative Study of Stream Reasoning Engines

Nathan Gruber(✉) and Birte Glimm

Ulm University, Helmholtzstraße 16, 89081 Ulm, Germany
nathan.gruber@tum.de, birte.glimm@uni-ulm.de

Abstract. The diverse research efforts in recent years in the area of stream reasoning (SR) led to a wide range of SR engines. However, the lack of standardization and the diverse choices in SR (e.g., tuple-driven vs. time-driven engines, streaming all results vs. newly derived ones, ...) mean that real comparability among the engines is hardly given. A first step towards achieving comparability and standardization is the RSP-QL model, implemented in the RSP4J framework, which allows for describing and formalizing the semantics of SR engines. To further advance the state of the art in comparative research of stream reasoning, we present the results of a survey to quantify the in-use importance of several key performance indicators (KPIs) and features and compare SR engines along these KPIs with the CityBench and the CSRBench oracle. Our analysis shows that the two RSP4J implementations C-SPARQL2.0 and YASPER outperform the well-known C-SPARQL implementation in terms of performance and configurability. Our comparison against a naive SR extension of the incremental reasoning engine RDFox shows that SR engines still have potential for improvement. To avoid a costly integration of engines into several different benchmarking environments, we finally present a unifying interface, already aligned with the CityBench and CSRBench, for benchmarking SR engines.

Keywords: Stream Reasoning · RSP4J · RDFox · C-SPARQL · CityBench · CSRBech · Benchmarking Interface

1 Introduction

Research in the area of stream reasoning (SR) has gained popularity in recent years [10] because the fields of application are vast, reaching from smart cities [11] over industry 4.0 scenarios [12] to the internet of things [22]. Stream reasoning aims at making sense of dynamic data streams combined with static background knowledge in real-time. To meet the requirements necessary for reaching this vision, researchers theoretically investigated the area of stream reasoning, suggested approaches and models that tackle some requirements, and built stream reasoning engines, mainly as proof of concept implementations, to test their approaches and compare them in practice. Examples for such stream reasoning engines are C-SPARQL [4], SPARQL$_{STREAM}$ [6], and CQELS [16].

C. Pesquita et al. (Eds.): ESWC 2023, LNCS 13870, pp. 21–37, 2023.
https://doi.org/10.1007/978-3-031-33455-9_2

Since these engines have been developed individually by different research groups, each having their own ideas and no binding standard being established, there are significant differences in the engines' behavior, performance, and functionality. Multiple query languages have been proposed as well, which fostered the breadth of research but hindered a fair comparison between stream reasoning engines. In order to identify and name differences in the operational semantics of existing engines, Botan et al. [5] came up with the descriptive SECRET model for stream reasoning engines that was then extended into a unifying standard model for the semantics of SR engines, the RSP-QL model [8]. The recently published RSP4J framework [23] implements this standard and proposes prototype engines. In this paper, we present the following contributions:

- While it is broadly recognized that benchmarking fosters research and applications by helping to identify superior techniques and best practices, it is less clear which measurable KPIs are the most relevant ones in practice. We empirically evaluate the in-use importance of several KPIs and features through a survey, which gives a clearer context for benchmark results.
- We analyze four stream reasoning engines theoretically and compare their performance regarding several KPIs using common SR benchmarks. The engines C-SPARQL2.0[1] and YASPER [24] are based on the newly introduced RSP4J framework, while another one is the commonly known baseline implementation C-SPARQL [4]. The last one is the high-performance incremental reasoning engine RDFox [17], which we extended for its use as SR engine.
- In order to avoid the time-consuming task of integrating one SR engine at a time into different benchmarking environments, we developed a flexible, unifying interface for benchmarking SR engines. Two popular benchmarks, namely CityBench and CSRBench, are already aligned with the interface and can, hence, directly be used with any SR engine adapting the interface.

The paper is structured as follows: Sect. 2 introduces the definitions that are necessary for understanding what follows. Section 3 gives an overview of related work and, in Sect. 4, we introduce the considered engines and provide a theoretical categorization. In Sect. 5, we showcase and discuss the results of our survey as well as of our benchmarking efforts before we present the unifying interface for SR benchmarks in Sect. 6. Section 7 sums up our findings and gives an outlook on open issues that should be addressed in future research.

2 Preliminaries

We assume interested readers to be familiar with the basics of the RDF data model [19] and the SPARQL query language [2]. In the following, we focus on RDF Stream Processing (RSP) aspects, i.e., on extensions of RDF and SPARQL for dealing with the continuous processing of (RDF) data streams.

Stream reasoning engines work on (static) RDF graphs in combination with dynamic RDF streams. Those streams are RDF triples combined with a monotonously increasing timestamp that indicates the arrival time of the triples:

[1] https://github.com/streamreasoning/csparql2.

Definition 1 (RDF Streams). *Let $t_0, \ldots, t_n$ be RDF triples and $\tau_0, \ldots, \tau_n$ timestamps such that, for each $0 \leq i < j \leq n$, $\tau_i < \tau_j$, then the sequence $(\langle t_0, \tau_0 \rangle, \ldots, \langle t_n, \tau_n \rangle)$ is an RDF* stream*.*

Note that we can see an RDF stream $(\langle t_0, \tau_0 \rangle, \ldots, \langle t_n, \tau_n \rangle)$ w.r.t. a current timestamp τ_c, $\tau_0 \leq \tau_c \leq \tau_n$, such that the sub-sequence $(\langle t_0, \tau_0 \rangle, \ldots, \langle t_i, \tau_i \rangle)$, $\tau_0 \leq \tau_i < \tau_c$, consists of past and the sub-sequence $(\langle t_j, \tau_j \rangle, \ldots, \langle t_n, \tau_n \rangle)$, $\tau_c < \tau_j \leq \tau_n$ consists of future (timestamped) triples.

Most SR engines work on snapshots of the data streams, so-called *windows*, which consist of a sub-sequence of the streamed data w.r.t. some point in time.

Definition 2 (Windows). *Given an RDF stream $(\langle t_0, \tau_0 \rangle, \ldots, \langle t_n, \tau_n \rangle)$ and a current timestamp τ_c, $\tau_0 \leq \tau_c \leq \tau_n$, a* physical window *of* (window) size *$w \in \mathbb{N}$ is the sub-sequence $(\langle t_i, \tau_i \rangle, \ldots, \langle t_c, \tau_c \rangle)$ of $(\langle t_0, \tau_0 \rangle, \ldots, \langle t_n, \tau_n \rangle)$ such that $\#\{\tau_i, \ldots, \tau_c\} = w$, i.e., the window consists of the last w triples.*

Given an RDF stream $(\langle t_0, \tau_0 \rangle, \ldots, \langle t_n, \tau_n \rangle)$, a starting time τ_0, a window size $w \in \mathbb{N}$, and a step size *$s \in \mathbb{N}$ $(s \leq w)$, the i^{th}* logical window *W_i opens at $\tau_o = \tau_0 + i * s$, closes at $\tau_c = \tau_o + w$ and contains the (timestamped) triples $\{\langle t, \tau \rangle \mid \tau_o \leq \tau < \tau_c\}$. A logical window is called a* tumbling window*, if $s = w$, and it is called a* sliding window *if $s < w$.*

Note that in practice, one might also consider initial physical windows that contain less than w triples, whereas the current definition considers the first window to be defined only when w triples are available at the current point in time. Note further that the contents of tumbling windows are always non-overlapping, whereas sliding windows have an overlap.

Since most SR engines perform reasoning and query answering on the (static) windows, we distinguish three types of necessary operators: *Stream-To-Relation (S2R)*, *Relation-to-Relation (R2R)*, and *Relation-to-Stream (R2S)*, where S2R operators typically perform some kind of windowing on data streams, R2R operators usually process SPARQL-like queries that produce static variable bindings (i.e., again relational data), and R2S operators transform these bindings back into data streams [3]. Regarding R2S operators, we distinguish RStreams, which emit the current solution mappings, IStreams, which emit the difference between the current and the previous solution mappings, and DStreams, which emit the difference between the previous and the current solution mappings.

Most SR query languages are extensions of SPARQL that additionally allow for setting the necessary information to execute SPARQL-like queries continuously over given data streams. An example is given in Listing 1.1, which registers a query as an RStream (Line 3) and specifies windows over which patterns are to be executed (Lines 5 and 7).

Window-based querying opens up several choices in terms of query execution semantics. The SECRET model [5] allows for characterizing these choices along four complementary dimensions: **S**cop**E**, **C**ontent, **RE**port, and **T**ick. Given a query's window parameters, *ScopE* defines the time interval for the active window. *Content* then specifies the elements of a stream that are in scope. *REport* states under what conditions those window contents become visible to the query

```
PREFIX ses: <http://www.insight-centre.org/dataset/SampleEventService#>
PREFIX ssn: <http://purl.oclc.org/NET/ssnx/ssn#>
REGISTER RSTREAM <q1> AS
SELECT ?obId1
FROM NAMED WINDOW <w1> ON ses:AarhusTrafficData182955 [RANGE PT3S STEP PT1S]
FROM <http://localhost/WebGlCity/RDF/SensorRepository.rdf>
WHERE { WINDOW <w1> { ?obId1 ssn:observedBy ses:AarhusTrafficData182955 . } }
```

Listing 1.1. An example (streaming) query

processor for evaluation and result reporting. Possible report strategies include the *window close* strategy, where results are reported, when a window closes, *periodic* (time-based) reporting, and reporting when the current window contains changed content (*content change*), or is non-empty (*non-empty content*). Finally, *Tick* (aka “window state change” or “window re-evaluation”) models what drives an SR engine to take action on its input, which can, for example, be tuple-driven (when a triple arrives) or time-driven.

3 Related Work

To enable meaningful benchmarking in the area of stream reasoning, descriptive models are needed that can characterize the differences between existing engines and, thus, raise the comparability. On the other hand, benchmarks should push the systems to their limits in various respects and automatically measure the corresponding KPIs.

Correctness and Comparability. Only if the SECRET primitives of different engines are aligned and well understood, a fair comparison can be made. The SECRET framework is the basis for the RSP-QL model [8], which aims at unifying the semantics of SR engines. RSP-QL extends the SECRET model with data types like time-varying graphs, instantaneous graphs, and R2S operators and, thus, allows for a formal definition of correctness in SR systems. The correctness of SR engines can be checked automatically using the CSRBench [9], which comes with a configurable oracle that compares the answers of an SR engine with expected answers w.r.t. its SECRET primitives and for different start times. The RSP4J framework [23] is the first RSP-QL compliant Java API designed to facilitate the building of new SR engines and to foster the comparability among these systems.

Benchmarking. Scharrenbach et al. consider inference support over background knowledge or correct and efficient time modeling as essential properties of stream processing systems and, against this background, propose seven commandments for effective benchmarking of stream processing systems [21]. The benchmarks should challenge the systems in different dimensions including load balancing, various joins and aggregates, as well as the usage of various types of background knowledge. These proposals and the CityBench [1], which is based on real sensor data collected within the CityPulse project, combined with imaginary

Table 1. Properties of the considered SR engines

C-SPARQL	**C-SPARQL2.0 & YASPER**	**RDFox**
Scope		
physical & logical windows start time cannot be set	logical windows start time can be set	logical windows start time cannot be set
Content / RSP-QL Dataset		
content merged into default graph	individually named windows	content merged into default graph
Report		
window close & non-empty content	configurable	periodic
Tick		
time-driven	configurable	time-driven (configurable interval)
R2S Operator		
RStream empty relations are transmitted	configurable empty relations are not transmitted	RStream empty relations are not transmitted

movement data, are the basis of our work. The CityBench allows for various configurations, from the input stream rate to different sized background data files, over queries with variable numbers of input streams, to a configurable number of queries to be executed in parallel. Meanwhile, the CityBench measures the latency, memory consumption, and completeness of the considered engines.

Because it is time-consuming to integrate several SR engines into multiple benchmarks individually, Tommasini et al. proposed approaches to unify parts of the SR engines with Heaven [25] and RSPLab [26] and Kolchin et al. developed the YABench as an extensive benchmarking framework with multiple supported KPIs [14]. In addition, the results of some practical comparisons of the performances of existing engines have been published, e.g., comparing the performance of C-SPARQL and CQELS [7,20].

The HOBBIT platform[2] aims at providing a general, distributed, open-source, evaluation platform for semantic technologies. Due to its generality, the platform is more complex than a dedicated SR benchmark, while it is not targeting SR intricacies such as aligning the systems along the SECRET primitives.

4 Stream Reasoning Engines

In the following, we introduce the evaluated SR engines and compare them with regard to their SECRET and RSP-QL primitives (see Sect. 2 for the necessary definitions). Table 1 summarizes the categorization of the engines.

C-SPARQL. Continuous SPARQL (C-SPARQL) was introduced in 2010 by Barbieri et al. [4] as a query language extension for SPARQL. It came along with an open-source proof-of-concept Java implementation, the C-SPARQL engine,

[2] https://project-hobbit.eu/.

which is still a common baseline implementation. The C-SPARQL engine is provided as an open-source Java project on GitHub.[3]

C-SPARQL2.0 and YASPER. C-SPARQL2.0[4] and YASPER [24] are the first two prototype engines based on RSP4J, provided as Java open-source projects. While C-SPARQL2.0 uses Esper[5] for windowing and Jena[6] for querying, YASPER is a from-scratch implementation within RSP4J. C-SPARQL2.0 can be used to evaluate the performance of the RSP4J framework whereas YASPER, with its high degree of abstraction, is built for teaching. RSP4J-based engines allow for configuring their SECRET primitives and as they are RSP-QL compliant, using RSP-QL as a query language, they are more expressive than, for example, C-SPARQL due to the naming of the time-varying graphs.

RDFox. RDFox [17] is a highly-scalable, parallelized in-memory RDF store, which is currently commercially licensed and maintained by Oxford Semantic Technologies.[7] RDFox is not a stream reasoning engine per se, but it supports incremental (datalog) reasoning, has an extensive query support and shows an excellent performance. Hence, a comparison with RDFox gives an interesting perspective on the performance of dedicated SR engines.

For extending RDFox into an SR engine, we used similar SECRET primitives as those of C-SPARQL and implemented logical windowing. We parse queries in two parts: first, the information about the static/dynamic data and the windows is read, before the actual SPARQL query is parsed. We deliberately store the static and streamed data in the same data store (within the default graph) this allows for processing all data at the same time and perform reasoning on it. Since C-SPARQL follows a time-driven tick and a window-close report strategy, which is no different from a time-driven report strategy except for the first window, we implemented a time-driven tick and report strategy. We update the data store at the periodic step size interval of every stream (*report*) and evaluate the query every 15 milliseconds (*tick*).[8] Implemented optimizations include the adjustment of the tick interval and the adding of an offset to the data store updates depending on the number of concurrent queries, as well as the aggregation of the streamed data prior to the execution of data store updates. Since the internal dictionary for recording resources in RDFox grows with newly arriving triples, we regularly refreshed the data store (exported and reimported the triples) depending on the relative size of the dictionary. This prevents the RDFox server from growing linearly during the runtime and only has a negligible impact on the latency. We refer interested readers to GitHub[9] for the complete implementation details.

[3] https://github.com/streamreasoning/CSPARQL-engine.

[4] https://github.com/streamreasoning/csparql2.

[5] https://www.espertech.com/esper/.

[6] https://jena.apache.org/.

[7] https://www.oxfordsemantic.tech/.

[8] The tick interval of 15 milliseconds was chosen experimentally as it was a good trade-off between low latency and not putting too much load on the engine.

[9] https://github.com/SRrepo/CityBench-CSPARQL-RDFox/tree/master/src/org/java/aceis/utils/RDFox.

5 Evaluation

In this section, we present the results of our survey on the in-use importance of features and KPIs of SR engine as well as the benchmarking results.

5.1 Features and Key Performance Indicators

We start by introducing a list of features and KPIs which are the structured sum of previous publications and community discussions [1,9,20]. We further provide a short theoretical comparison of the considered SR engines regarding each feature (see Table 2) while the presented benchmarking results cover the remaining KPIs.

Latency. The latency of a stream reasoning engine refers to the average amount of time between the input arrival and the output generation for every triple. This performance indicator can be measured within the CityBench testbed. All considered engines internally use a periodic/time-driven tick strategy, which is essential for a fair comparison. A fair comparison with CQELS [16], for example, would not be possible because CQELS instantly reacts to the arrival of each triple and, therefore, offers a better latency.

Memory Consumption. The memory consumption of a stream reasoning engine refers to the average amount of memory used by the engine during its runtime. The CityBench testbed provides a possibility to measure this performance indicator. However, it also impurifies the results because it does not distinguish between the memory consumption of the engine and the memory consumption of the testing environment (which, for instance, stores the processed answers). To give a finer-grained picture, we analyzed the memory consumption during every execution in detail.

Completeness. The completeness of a stream reasoning engine refers to the percentage of correctly processed input triples. In our setup of the CityBench testbed, completeness is measured by setting the number of unique observations captured by the SR engine ($\#CO$) in relation to the number of unique observations produced by the test bed generator ($\#PO$), i.e., we compare $\#CO/\#PO$.

Maximum Throughput. An SR engine's maximum throughput characterizes the maximum amount of RDF triples that the engine can handle per time unit. This performance indicator cannot be measured automatically by any (existing) benchmarking environment because the maximum throughput depends, for example, on the complexity of the query, the size of the static background data, and the number of concurrent queries. The CityBench, however, allows for configuring the frequency and rate of input streams. In the following, we compare the latency, memory consumption, and completeness of the engines as a function of the input rate of the streams. Especially the dependencies between the input rate and the latency/completeness allow for making statements about the maximum throughput of the engines.

Correctness/Approximation Quality. We refer to correctness in the context of stream reasoning engines in terms of RSP-QL correctness [8]. To validate the functional correctness of the engines with respect to their execution semantics, we used the CSRBench oracle.

Table 2. Features of the considered SR engines

C-SPARQL	**C-SPARQL2.0 & YASPER**	**RDFox**
Support of Background Data		
supported	supported	supported
(RDFS-)Reasoning / Inference Support		
RDF entailment	OWL 2 entailment	OWL 2 entailment
Distributed Streams		
supported	supported	supported
Distribution Computation		
not supported	not supported	parallelization/distribution across physical cores
Guaranteed Performance Level		
not supported	not supported	not supported
Intern Mode of Operation		
	see Table 1	
Special Query Language Features / Expressiveness		
timestamp function	naming of streams	extensive datalog reasoning
Maintenance and Further Development		
research group maintained	research group maintained	commercial software
no active development	under active development	under active development

Support of Background Data. This feature refers to an SR engines' ability to process queries with respect to static background data and streamed data.

(RDFS-)Reasoning/Inference Support. Stream reasoning engines generally allow for reasoning under a specific entailment regime. This means, they can entail new facts from given data and knowledge [13].

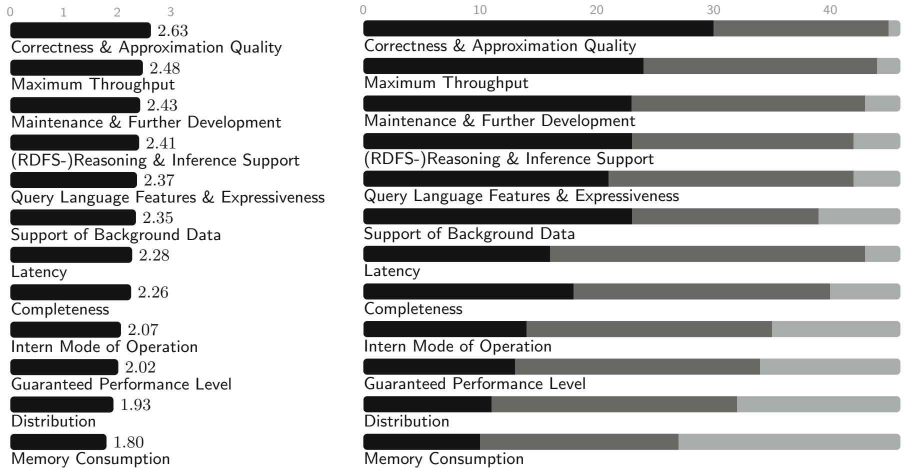
Distribution. The distribution of an SR engine can either refer to its ability to process distributed data streams or its ability to work on multiple physical machines in parallel.

Guaranteed Performance Level. If an SR engine supports a guaranteed performance level, it allows for configuring a maximum time interval between the initiation and the answering of a query (e.g., 50 ms). The answers might be incomplete or approximated, but in time.

Intern Mode of Operation. For some applications, the intern mode of operation (e.g., the supported windowing technique, the tick strategy, or the supported R2S operators) of an SR engine might be relevant.

Special Query Language Features/Expressiveness. We consider features that are rarely supported by SR engines and suit a particular application use case, as special features. Expressiveness refers to the complexity of queries that are supported by the respective query language of an engine.

Maintenance and Further Development. The level of maintenance, support, and the regularity of updates for an engine can be important factors for users of SR engines.



(a) Survey Results: On Feature Importance in Applications

(b) Survey Results: Detailed Assessments very important = black, important = dark gray and not so important = light gray

Fig. 1. Survey Results

5.2 Survey Results

Given the multitude of features and KPIs, the question arises as to which of them are particularly important for many real-world applications. To address this question, we conducted a survey with 46 developers, who worked on applications that internally use an SR engine. We asked them to assess the importance of each feature and performance indicator for their application on a scale of 1 (*not so important*), 2 (*important*), and 3 (*very important*). The results are presented in Fig. 1 with full details available on GitHub.[10]

The results indicate, that the correctness of a stream reasoning engine is essential to almost any application. It is, therefore, very important for an SR engine to pass the CSRBench (or YABench [14]). The survey results also suggest that many developers consider the maximum throughput of an SR engine to be more important for their application than the engines' latency. Furthermore, developers need SR engines to be reliable and maintained to benefit from their usage. The distribution (to multiple physical machines) and the memory consumption were rated as least important features/KPIs.

5.3 Benchmarking Setup

We performed all our experiments with the CityBench on a Lenovo ThinkPad T480 with 8 Intel(R) Core(TM) i5-8350U CPUs clocked at 1.70GHz and 8 GB RAM. Note that due to space limitations, we can only present a representative

[10] https://github.com/SRrepo/SurveyResults.

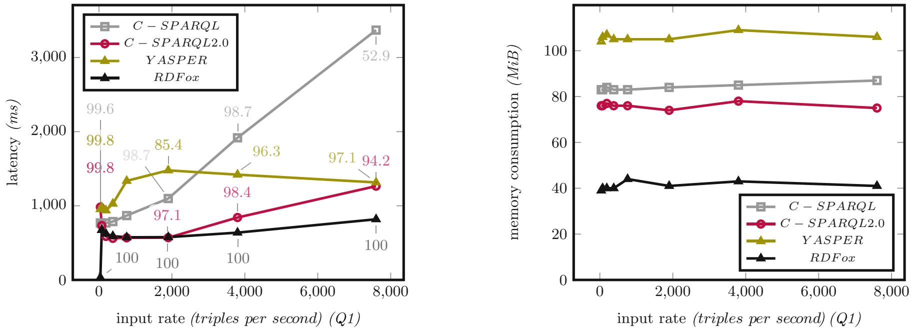


Fig. 2. Change of KPIs with a varying input rate

sample of the CityBench results below. All the setups (CityBench and CSR-Bench) and complete results are available in GitHub.[11] We used RDFox 5.4 and left the number of allowed threads to the default, which is the number of logical cores. For all other engines, we used their latest versions from GitHub.[12]

To improve the runtime efficiency, we performed two warm-up runs that lasted 130 s each before we started the experiments, which lasted 10 min or until all the available data was streamed.

5.4 CityBench Results

To examine the performance of the engines from different angles and to push the systems to their capacity limits in various ways, we used four scalability factors: the input rate, the number of concurrent queries (duplicity), the size of background data, and the number of parallel input streams. The following diagrams show the changes in latency, memory consumption, and completeness of the engines as a function of these scalability factors. The completeness is marked as a percentage next to the latency.

We used *Query 1* of the CityBench for the presented experiments with the input rate and duplicity and variants of *Query 1* which use larger background files for the experiments with varying background data. For the experiments with an increasing number of input streams, we used variants of *Query 10*. Note that the complexity of the chosen queries is rather low as they neither include calculations, filters, and aggregations nor UNION and OPTIONAL, which are computationally more complex operators [18].

Input Rate. The results in Fig. 2 show that the RDFox-based engine can handle large data streams without problems using its incremental maintenance

[11] https://github.com/SRrepo/.

[12] C-SPARQL2.0 commit number: f682cdc427d85594b39f9b4aa8d86e04833c8368,
YASPER commit number: aea74443955e1ab3b95de7b0ef65f7c1dbd51d08,
C-SPARQL commit number 4be27dd5ca23550da6bf7fb4e3420b0eb75132f0.

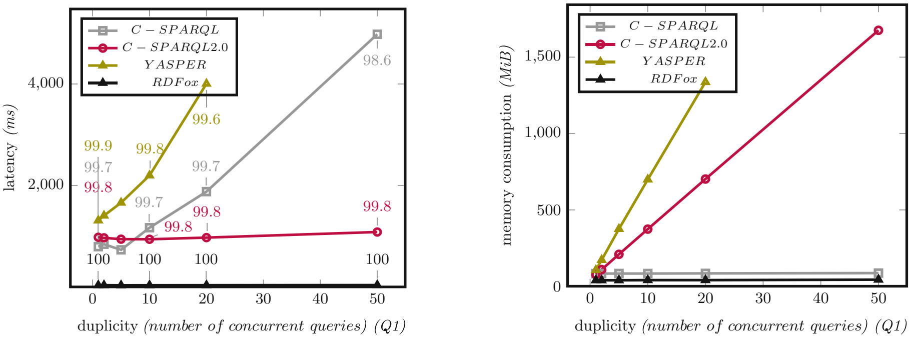


Fig. 3. Change of KPIs with a varying number of concurrent queries

algorithm. The implementation can be seen as a baseline in terms of completeness because it does not drop input triples when it faces an overload and is built in a way that no input is missed at the boundaries of a window. Meanwhile, the system's memory consumption is minimal. The RSP4J-based engines can also handle large amounts of input triples without losing much in terms of completeness. Their completeness decreases in dependence of how many input triples get lost at the boundaries of a window. C-SPARQL, on the other hand, successfully processes the query with an increased latency at a high completeness level up to around 5500 triples per second before its completeness suddenly collapses.

Duplicity. The results for concurrently executed queries are shown in Fig. 3. If the number of queries to be executed in parallel rises, the SR extension of RDFox increases its tick interval and adds an offset. Because the same query is executed multiple times and the results are evaluated centrally in the CityBench testbed, no significant changes in performance can be detected. If different queries were executed, the latency would increase to a uniform but overall higher level, depending on the tick interval, while the completeness would remain the same. The memory consumption would rise because multiple data stores were needed. With the RSP4J-based engines, significant memory consumption can be observed because these engines cache the static data for each query individually. The latency of C-SPARQL2.0 remains almost constant, whereas YASPER was not able to execute 50 queries in parallel because the available system memory was exceeded. C-SPARQL shows an increasing latency, while its memory usage remains constant in the trade-off between memory consumption and latency.

Background Data. Figure 4 shows the results of our experiments with background data sets of different sizes. The RDFox extension once again exhibits a low latency because its periodic report is initiated exactly after the first receipt of input triples. Only the latency of C-SPARQL is affected negatively by an increased amount of background data. On the other hand, the results indicate that RDFox manages large amounts of data in a very memory-efficient way compared to the conventional SR engines.

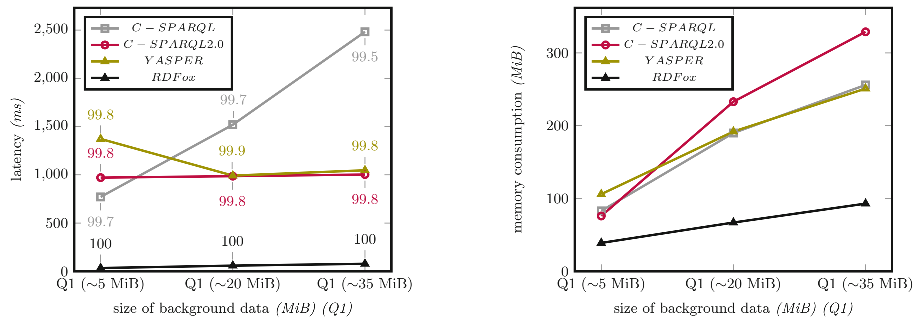


Fig. 4. Change of KPIs with a varying size of background data

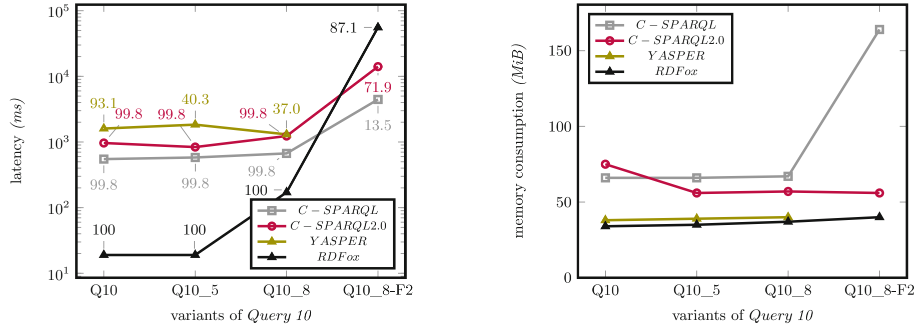


Fig. 5. Change of KPIs with a varying number of input streams

Number of Input Streams. Finally, our findings in combination with different numbers of input streams are depicted in Fig. 5. We used *Query 10*, *Query 10_5*, and *Query 10_8* of the CityBench and increased the frequency of *Query 10_8* to 2 (Q10_8-F2). These queries do not only test the engines' capabilities to process multiple input streams, but also their ability to handle a large number of answers because *Query 10_8* produces 3^8 (6^8) output bindings per second at a frequency level of 1 (2). The results show that RDFox and C-SPARQL2.0 are able to handle a large number of query answers per time unit while C-SPARQL and YASPER face an overload.

Discussion. Generally, our performance measurements for C-SPARQL are consistent with most previous work [1,15,20] though the presentation of our results differs from previous publications as we directly relate completeness to latency. Thus, it is noticeable when stable latency is measured for an engine simply because it no longer processes all results correctly. The presented results indicate that RDFox, which is (one of) the most efficient and highly optimized incremental reasoning engines, is also competitive when being used as SR engine. Furthermore, the empirical results illustrate the potential for improvement that

Table 3. RSP-QL correctness of SR engines using the CSRBench oracle

	C-SPARQL	**C-SPARQL2.0**	**YASPER**	**RDFox**
Query 1	✓	✓	✓	✓
Query 2	✓	✓	✓	✓
Query 3	✓	✓	✓	✓
Query 4	✓	✓	–[a]	✓
Query 5	×	×[b]	×[b]	✓
Query 6	✓	✓	✓	✓
Query 7	✓	✓	✓	✓

[a] YASPER does not yet support the AVG-function, which is necessary for *Query 4*.
[b] For a sliding window with a window size of 5 time units and a step size of 1 time unit and assuming an RStream, C-SPARQL2.0 and YASPER are expected to return every answer five times. While most answers are indeed returned five times, some are only returned four times. We conjecture that this is due to internal temporal imprecisions introduced by a processing overhead.

still exists in the prototypical and not fully optimized implementations of the RSP4J framework.

5.5 CSRBench Results

Since our survey underlines the importance of the correct functioning of SR engines, the results of the CSRBench take on special significance. Table 3 shows the results of our experiments with the CSRBench oracle. All queries but Query 5 of the CSRBench use tumbling windows, for which the periodic and the window-close report strategies are equivalent (see Sect. 2). While C-SPARQL's answers do not match the expected answers for Query 5 because it reports the first window before it closes [9], the answers of our RDFox extension were accepted, even though the engine uses a periodic report strategy which is not supported by the oracle. This is because RDFox does not return empty answers and the first open windows do not contain any data that produces answers.

6 A Unifying Interface for Stream Reasoning Benchmarks

As it is very time-consuming to integrate one SR engine at a time into multiple benchmarking environments, we suggest a unifying interface for SR benchmarks that should speed up future benchmarking. The proposed interface is divided into three consecutive phases: the *initialization* (see Fig. 6), the *processing* (see Fig. 7), and the *evaluation* phase (see Fig. 8). Some parameters such as the *engine name* and specific *benchmark parameters* are required. Optional parameters include the *RDF serialization format* in which the data is streamed, the *configuration URL*, the *answers URL*, the *query language* of the engine, and the *waiting time* after the streams end. Reasonable defaults for the parameters

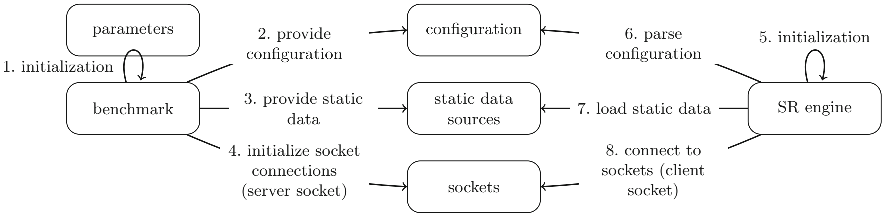


Fig. 6. Sequence and steps of the initialization phase

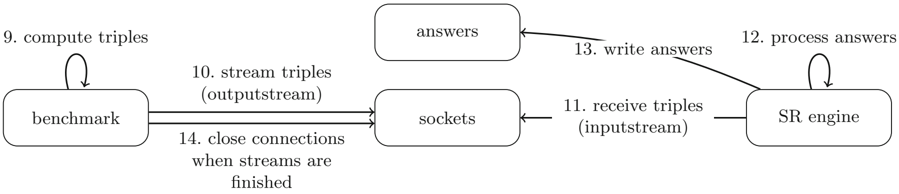

9. compute triples
benchmark
10. stream triples (outputstream)
14. close connections when streams are finished
answers
sockets
13. write answers
11. receive triples (inputstream)
12. process answers
SR engine

Fig. 7. Sequence and steps of the processing phase

are foreseen and can be found in GitHub.[13] The interface comes along with several advantages:

Alignment of CityBench and CSRBench. We already aligned the CityBench[14] and the CSRBench[15] with the interface. Using these proof-of-concept implementations, we were able to reproduce the results presented in the last section with an aligned C-SPARQL[16], and central classes of these projects can easily be used for the alignment of further benchmarks.

Speed Up. The unifying interface speeds up benchmarking for developers of benchmarks as well as SR engines. For simplicity, let us assume that three SR

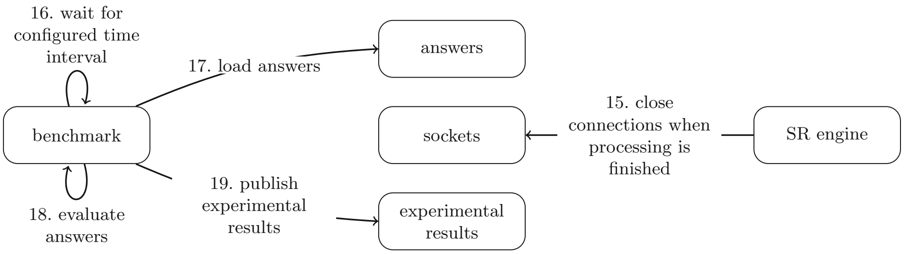

16. wait for configured time interval
benchmark
18. evaluate answers
17. load answers
19. publish experimental results
answers
sockets
experimental results
15. close connections when processing is finished
SR engine

Fig. 8. Sequence and steps of the evaluation phase

[13] https://github.com/SRrepo/CSRBench-Aligned/blob/master/Parameters.md.

[14] https://github.com/SRrepo/CityBench-Aligned.

[15] https://github.com/SRrepo/CSRBench-Aligned.

[16] https://github.com/SRrepo/CSPARQL-Running-Example-For-Unifying-Interface.

engines and three benchmarks exist. Currently, if a new SR engine is developed and should be benchmarked, it has to be integrated into all three benchmarks individually. Since the developer first has to understand each benchmark's functionality, this process takes at least one week per benchmark, according to our experience. If all benchmarks were aligned with the interface, the developer would only have to write one single wrapper that implements the interface for the engine. Since dealing with the functionality of each individual benchmark is not needed any more and large parts of our C-SPARQL example can be adopted, this process probably takes less than one week. Analogously, the development time of a new benchmark can be significantly reduced by using the interface. In agreement with the developers of RSP4J, a standardized RSP4J runner component could even be used to automatically test every RSP4J-based engine with all aligned benchmarks, regardless of its implementation details.

Engine Independence. Another advantage of the interface is that the SR engine runs independently of the benchmark in a separate process. This allows for measuring, e.g., the memory consumption more cleanly and, by using sockets, standardized RDF formats, and JSON for the communication between benchmark and engine, we enable a smooth benchmarking of SR engines that are not written in Java or already compiled.

Expandability. As the engines store all data in a time-annotated fashion and the benchmark evaluates the performance retrospectively, future benchmarks can easily introduce new KPIs.

7 Conclusions

In this work, we empirically demonstrated RSP4J to be sound and performant. We also highlighted its advantages over C-SPARQL in the prototype implementations C-SPARQL2.0 and YASPER. Furthermore, this work reveals how a high-performant SR engine can be built on top of the incremental reasoning engine RDFox. The presented survey highlights the importance of functional correctness in SR engines for real-world applications and the unifying interface for SR benchmarks forms the basis for simplified future benchmarking.

Future research could, on the one hand, extend the RSP4J framework with a unified benchmark-runner and, on the other hand, further optimize RSP4J, e.g., by introducing algorithmic optimizations on its operators. In addition, it is worth supplementing this comparison with other engines, especially from the EP area, and building a standardized database for benchmarking results of SR engines. More generally, it is certainly useful to test other programming languages than Java in the context of SR engines. Last but not least, the incompleteness of SR engines endangers their use in real-world applications. There is an urgent need to explore theoretically and in practice how the completeness and reliability of SR engines can be raised. If an SR engine drops input to prevent an overload, for instance, it will be crucial to investigate how priorities can be assigned to certain inputs or how the dropping of input can be realized in a way that affects the query results as little as possible and, ideally, in a quantifiable manner.

References

1. Ali, M.I., Gao, F., Mileo, A.: CityBench: a configurable benchmark to evaluate RSP engines using smart city datasets. In: Arenas, M., et al. (eds.) ISWC 2015. LNCS, vol. 9367, pp. 374–389. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-25010-6_25
2. Aranda, C.B., et al.: SPARQL 1.1 overview. W3C recommendation, W3C (2013). https://www.w3.org/TR/2013/REC-sparql11-overview-20130321/
3. Arasu, A., et al.: STREAM: the Stanford data stream management system. In: Data Stream Management. DSA, pp. 317–336. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-540-28608-0_16
4. Barbieri, D.F., Braga, D., Ceri, S., Valle, E.D., Grossniklaus, M.: C-SPARQL: a continuous query language for RDF data streams. Int. J. Semant. Comput. **4**(01), 3–25 (2010)
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., et al. (eds.) ISWC 2010. LNCS, vol. 6496, pp. 96–111. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-17746-0_7
7. Dao-Tran, M., Beck, H., Eiter, T.: Contrasting RDF stream processing semantics. In: Qi, G., Kozaki, K., Pan, J.Z., Yu, S. (eds.) JIST 2015. LNCS, vol. 9544, pp. 289–298. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-31676-5_21
8. Dell'Aglio, D., Della Valle, E., Calbimonte, J.P., Corcho, O.: RSP-QL semantics: a unifying query model to explain heterogeneity of RDF stream processing systems. Int. J. Semant. Web Inf. Syst. **10**, 17–44 (2014). https://doi.org/10.4018/ijswis.2014100102
9. 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.) Semantic Web Conference, pp. 326–342. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-41338-4_21
10. Dell'Aglio, D., Della Valle, E., van Harmelen, F., Bernstein, A.: Stream reasoning: a survey and outlook. Data Sci. **1**(1–2), 59–83 (2017)
11. D'Aniello, G., Gaeta, M., Orciuoli, F.: An approach based on semantic stream reasoning to support decision processes in smart cities. Telemat. Inform. **35**(1), 68–81 (2018). https://doi.org/10.1016/j.tele.2017.09.019. https://www.sciencedirect.com/science/article/pii/S0736585317304768
12. Giustozzi, F., Saunier, J., Zanni-Merk, C.: Abnormal situations interpretation in industry 4.0 using stream reasoning. Procedia Comput. Sci. **159**, 620–629 (2019). https://doi.org/10.1016/j.procs.2019.09.217. https://www.sciencedirect.com/science/article/pii/S1877050919314012. Knowledge-Based and Intelligent Information & Engineering Systems: Proceedings of the 23rd International Conference KES2019
13. Glimm, B., Ogbuji, C.: SPARQL 1.1 entailment regimes. W3C recommendation, W3C (2013). https://www.w3.org/TR/2013/REC-sparql11-entailment-20130321/
14. Kolchin, M., Wetz, P., Kiesling, E., Tjoa, A.M.: YABench: a comprehensive framework for RDF stream processor correctness and performance assessment. In: Bozzon, A., Cudre-Maroux, P., Pautasso, C. (eds.) ICWE 2016. LNCS, vol. 9671, pp. 280–298. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-38791-8_16

15. Lachhab, F., Bakhouya, M., Ouladsine, R., Essaaidi, M.: Performance evaluation of linked stream data processing engines for situational awareness applications. Concurr. Comput. Pract. Exp. **30**(12), e4380 (2018)
16. 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., et al. (eds.) ISWC 2011. LNCS, vol. 7031, pp. 370–388. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-25073-6_24
17. Nenov, Y., Piro, R., Motik, B., Horrocks, I., Wu, Z., Banerjee, J.: RDFox: a highly-scalable RDF store. In: Arenas, M., et al. (eds.) ISWC 2015. LNCS, vol. 9367, pp. 3–20. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-25010-6_1
18. Pérez, J., Arenas, M., Gutierrez, C.: Semantics and complexity of SPARQL. In: Cruz, I., et al. (eds.) ISWC 2006. LNCS, vol. 4273, pp. 30–43. Springer, Heidelberg (2006). https://doi.org/10.1007/11926078_3
19. Raimond, Y., Schreiber, G.: RDF 1.1 primer. W3C note, W3C (2014). https://www.w3.org/TR/2014/NOTE-rdf11-primer-20140624/
20. Ren, X., Khrouf, H., Kazi-Aoul, Z., Chabchoub, Y., Curé, O.: On measuring performances of C-SPARQL and CQELS. arXiv preprint arXiv:1611.08269 (2016)
21. Scharrenbach, T., Urbani, J., Margara, A., Della Valle, E., Bernstein, A.: Seven commandments for benchmarking semantic flow processing systems. In: Cimiano, P., Corcho, O., Presutti, V., Hollink, L., Rudolph, S. (eds.) ESWC 2013. LNCS, vol. 7882, pp. 305–319. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38288-8_21
22. Su, X., Gilman, E., Wetz, P., Riekki, J., Zuo, Y., Leppänen, T.: Stream reasoning for the internet of things: challenges and gap analysis. In: Proceedings of the 6th International Conference on Web Intelligence, Mining and Semantics, WIMS 2016. Association for Computing Machinery, New York (2016). https://doi.org/10.1145/2912845.2912853
23. Tommasini, R., Bonte, P., Ongenae, F., Della Valle, E.: RSP4J: an API for RDF stream processing. In: Verborgh, R., et al. (eds.) ESWC 2021. LNCS, vol. 12731, pp. 565–581. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-77385-4_34
24. Tommasini, R., Della Valle, E.: Yasper 1.0: towards an RSP-QL engine. In: International Semantic Web Conference (Posters, Demos & Industry Tracks) (2017)
25. Tommasini, R., Della Valle, E., Balduini, M., Dell'Aglio, D.: Heaven: a framework for systematic comparative research approach for RSP engines. In: Sack, H., Blomqvist, E., d'Aquin, M., Ghidini, C., Ponzetto, S.P., Lange, C. (eds.) ESWC 2016. LNCS, vol. 9678, pp. 250–265. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-34129-3_16
26. Tommasini, R., Della Valle, E., Mauri, A., Brambilla, M.: RSPLab: RDF stream processing benchmarking made easy. In: d'Amato, C., et al. (eds.) ISWC 2017. LNCS, vol. 10588, pp. 202–209. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-68204-4_21