Heaven: a framework for systematic comparative research approach for RSP engines

Riccardo Tommasini, Emanuele Della Valle, Marco Balduini, Daniele Dell'Aglio

DEIB, Politecnico of Milano, Milano, Italy
riccado.tommasini@polimi.it, emanuele.dellavalle@polimi.it,
marco.balduini@polimi.it, daniele.dellaglio@polimi.it

Abstract. Benchmarks like LSBench, SRBench, CSRBench and, more recently, CityBench satisfy the growing need of shared datasets, ontologies and queries to evaluate window-based RDF Stream Processing (RSP) engines. However, no clear winner emerges out of the evaluation. In this paper, we claim that the RSP community needs to adopt a Systematic Comparative Research Approach (SCRA) if it wants to move a step forward. To this end, we propose a framework that enables SCRA for window based RSP engines. The contributions of this paper are: (i) the requirements to satisfy for tools that aim at enabling SCRA; (ii) the architecture of a facility to design and execute experiment guaranteeing repeatability, reproducibility and comparability; (iii) $\mathcal{H}$eaven – a proof of concept implementation of such architecture that we released as open source –; (iv) two RSP engine implementations, also open source, that we propose as baselines for the comparative research (i.e., they can serve as terms of comparison in future works). We prove $\mathcal{H}$eaven effectiveness using the baselines by: (i) showing that top-down hypothesis verification is not straight forward even in controlled conditions and (ii) providing examples of bottom-up comparative analysis.

1 Introduction

The Stream Reasoning (SR) [10] community agrees on the principle that Information Flow Processing approaches [9] and reasoning techniques can be coupled in order to reason upon rapidly changing information flows (for a recent survey see [18]). Currently, a W3C community group[1] is working towards the standardization of the following basic SR technologies:

- *RDF Streams*, which were introduced in [10], later on picked up in [16, 2] and approached as *virtual* RDF stream in [8].
- *Continuous extension of SPARQL* such as, in chronological order, C-SPARQL [5], SPARQL$_{stream}$ [8], CQELS-QL [16] and EP-SPARQL [2, 13].
- *Reasoning techniques* optimized for the streaming scenario such as, in chronological order, Streaming Knowledge Bases [27], IMaRS [6], a stream-oriented version of TrOWL [21], EP-SPARQL [2] and RDFox [20].

[1] http://www.w3.org/community/rsp/

These concepts gave birth to a new class of systems, collectively called RDF Stream Processing (RSP) engines, which proved SR feasibility and drew the attention of the community to their evaluation.

The goal of a domain specific benchmark is to foster technological progress by guaranteeing a fair assessment. For window-based RSP engines - the class of systems in the scope of this paper - RDF Streams, ontologies, continuous queries and performance measurements were proposed in LSBench [17], SRBench [28], CSRBench [11] and CityBench [1]. However, existing benchmarks showed no absolute winner and it is even hard to claim when an engine outperforms another.

In this paper, we argue that SR needs to identify the situations under which an engine A works better than an engine B, either quantitatively, i.e. focusing on a very aspect of the performance, or qualitative, i.e. classifying the behavior by observing the dynamics. This approach is known as Systematic Comparative Research Approach (SCRA) [15] and it is adequate when the complexity of the study subject makes it hard to formulate top-down hypothesis. Consequently, we formulate our research question as:

How can we enable a SCRA for window based RSP engines?

Our answer to this research question comprises:

1) A set of requirements to satisfy in order to enable a SCRA.

2) A general architecture for an RSP engine Test Stand; a facility that: (i) makes possible to design and systematically execute experiments; (ii) provides a controlled environment to guarantee repeatability and reproducibility of experiments; (iii) collects performance measurements while the engines are running; and (iv) allows us to comparatively evaluate the engines post-hoc.

3) $\mathcal{H}$eaven, a proof-of-concept implementation that we released open source[2], which improves the one proposed in [25] as explained in Section 5.

4) Two *baseline* RSP engine implementations that we propose as terms of comparison, since they are the kind of "simplified complex cases that combine known properties" advocated in SCRA [15] to highlight differences and similarities among real cases. They are released open source, too[2].

Demonstrating that the paper contributions positively answer our research question requires us to: (i) highlight that top-down hypothesis confirmation is not straight forward even in a controlled environment and when the observed RSP engines are as simple as the baselines are; (ii) prove $\mathcal{H}$eaven effectiveness by showing the relevance of the bottom-up analysis that it enables. To this extent, we first executed 14 experiments that involve the baselines as subjects and we showed that we cannot confirm the following two hypotheses, which have been already investigated top-down by the SR community [21, 26, 12, 20]:

Hp.1 Materializing from scratch the ontological entailment of the window content, each time it slides, is faster than the incremental maintenance of the materialization, when changes are large enough (e.g. greater than 10%)

[2] `https://github.com/streamreasoning/heaven`

Hp.2 Dually, if changes are small (e.g. less than 10%), the incremental maintenance of the materialization of the the window content ontological entailments, each time it slides, is faster than materializing it from scratch.

Then, we show some example of the bottom-up analysis that our approach enables. This part of our investigation is lead by four questions:

Q.a Qualitatively, is there a solution that always outperforms the others?
Q.b If no dominant solution can be found, when does a solution work better than another one?
Q.c Quantitatively, is there a solution that distinguishes itself from the others?
Q.d Why does a solution perform better than another solution under a certain experimental condition?

The remainder of the paper is organized as follows. Section 2 provides the minimum background knowledge required to understand the content of the paper. Section 3 presents the definition of RSP experiment and the requirements of a software framework to enable SCRA for RSP engines (i.e an architecture and at least a baseline). Section 4 shows the architecture of the RSP engine Test Stand and its workflow. Section 5 reports the implementation experience of $\mathcal{H}$eaven. Section 6 describes two RSP engines baseline. Section 7 shows the evaluation of $\mathcal{H}$eaven. Section 8 positions this paper within the state-of-the-art of RSP benchmarking. Finally, in Section 9, we come to conclusions.

2 Background

Stream Reasoning (SR) [10] is a novel research trend which focuses on combining Information Flow Processing (IFP) [9] engines and reasoners to perform reasoning on rapidly changing information. IFP engines are systems capable to continuously process **Data Streams**, that are potentially infinite sequences of events. The data streams considered in SR are the **RDF Streams**, where events are described by timestamped RDF data. For example, if at time 1, an event e_1 states that a MicroPost *:tweet*$_1$ is posted by *:alice*, while at time 5, a data item e_2 states that *:instagram_post*$_2$ is posted by *:bob*. The data stream $((e_1, 1), (e_2, 5))$ can be represented by the RDF Stream:

(:tweet$_1$ *sioc:has_creator :alice), 1*
(:instagram_post$_2$ *sioc:has_creator :bob), 5*

Most of the RSP query languages adopt a Triple-Based *RDF Stream Model*, i.e., events are represented by a single timestamped RDF statement. C-SPARQL [5], CQELS-QL [16] and SPARQL$_{stream}$ [8] are examples of this class of languages. However, a Graph-Based *RDF Stream Model* is possible and has been recently adopted by the RSP W3C group. An example of engine that adopts the Graph-Based RDF Stream model is SLD [4], that uses RDF graphs as a form of punctuation [23] to separate the events in the stream.

The IFP query languages and their processing methods can be grouped in two main membership classes:

- **Complex Event Processing (CEP) engines** that validate incoming primitive events and recognize patterns upon their sequences.
- **Data Stream Managements Systems (DSMSs)** [3] that exploit relational algebra to process portions of the data stream captured using special stream-to-relation (S2R) operators.

In this work, we target a class of Stream Reasoners known as window-based RDF Stream Processing (RSP) engines, inspired to DSMSs. The key operators of this class of engines are the S2R ones and, in particular, the sliding window operators. They allow to cope with the infinite nature of streams. Intuitively, a sliding window creates a view, named active window, over a portion of the stream that changes (slides) over time. **Time-based sliding windows** are sliding windows that create the active window and slide it accordingly to time constraints. They are defined by the width ω – i.e., the time range that has to be considered by the active window – and the slide β – i.e. how much time the active window moves ahead when it slides.

As advocated in the early works on Stream Reasoning [10, 27], the most simple approach to create a window-based RSP engine is pipelining a DSMS with a reasoner. For example, the C-SPARQL engine[3] uses Esper as DSMS and Jena as framework to manage RDF and reasoning over the window content. The C-SPARQL engine processes C-SPARQL queries under RDFS entailment regime[4]. For instance, the following C-SPARQL query asks to report every day (see STEP 15 min) the people mentioned during the last week in the stream of those who have published a paper (see RANGE 1d):

SELECT ?micropost
FROM STREAM <http://www.ex.org/socialStream> [RANGE 1d STEP 15 min]
WHERE {?micropost a :MicroPost}

The requested information is not explicitly stated in the RDF stream exemplified above, but being the range of the *sioc:has_creator* property a *:MicroPost*, an RDFS reasoner can deduce that :tweet$_1$ and :instagram_post$_2$ are of type :MicroPost and, thus, they belong to the answer of the query.

Other examples of window-based RSP engines are Morph$_{stream}$ [8] and CQELS [16], that adopt different approaches: the former is a native implementation of a window-based RSP engine, to achieve performance and adaptivity; the latter adopts an OBDA-like approach to rewrite the continuous query in a query to be DSMS and to be evaluated over a (non-RDF) data streams.

3 Notion of RSP Experiment and SCRA Requirements

An experiment is a test under controlled conditions that is made to demonstrate a known truth, examine the validity of a hypothesis[5], and that guarantees results

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

[4] http://www.w3.org/TR/sparql11-entailment/#RDFSEntRegime

[5] http://bit.ly/experiment-definition

reproducibility, *repeatability*, and *comparability*. This notion is not new in the RSP benchmarking state-of-the-art [1], but a formal definition is still missing.

We define an RSP Experiment as a tuple $< \mathcal{E}, \mathcal{T}, \mathcal{D}, \mathcal{Q}, \mathcal{K} >$ where:

- $\mathcal{E}$ is the RSP engine used as subject in the experiment;
- $\mathcal{T}$ is an ontology and any data not subject to change during the experiment;
- $\mathcal{D}$ is the description of the input data streams;
- $\mathcal{Q}$ is the set of continuous queries registered into $\mathcal{E}$; and
- $\mathcal{K}$ is the set of key performance indicators (KPIs) to collect.

The result of an RSP Experiment is a report $\mathcal{R}$ that contains the trace of the RSP engine processing for the entire duration of the experiment.

On these definitions and on the notions of *Comparability, Reproducibility* and *Repeatability*[6], we elicit the requirements for the architecture of a test stand that enables SCRA starting from an RSP Experiment.

Comparability refers to the nature of the experimental results $\mathcal{R}$ and their relationship with the experimental conditions. It requires that the test stand is [R.1] RSP engine agnostic, i.e. as long as the experimental conditions do not change, the results must be comparable, independently from the tested RSP engine. To this end, the report $\mathcal{R}$ has to record at least: (i) the data stream (or Stimulus) sent to $\mathcal{R}$; (ii) the application time of the Stimulus; (iii) the system time at which the Stimulus is sent; (iv) any KPIs to be measured before $\mathcal{E}$ processes the Stimulus; (v) the Response, if any, of $\mathcal{E}$ to the Stimulus; (vi) the system time at which the Response, if any, is sent; (vii) any KPIs to be measured after $\mathcal{E}$.

Reproducibility refers to measurement variations on a subject under changing conditions. It requires that the test stand is: [R.2] *data independent*, which means allowing the usage of any data stream and any static data (e.g., those proposed in [1, 11, 17, 28]); and [R.3] *query independent*, which means allowing the usage of any query from users' domains of interest (e.g., those proposed in [1, 11, 17, 28])

Repeatability refers to variations on repeated measurements on a subject under identical conditions. It requires that the test stand [R.4] minimizes the experimental error, i.e., it has to affect the RSP engine evaluation as less as possible and in a predictable way.

All these experiment properties together require the test stand to be [R.5] *independent from the measured key performance indicators (KPIs)*, i.e., the KPIs set has to be extensible. According to [22] a set of meaningful KPIs comprises: (i) *query-execution-latency* – the delay between the system time at which a Stimulus is sent to the RSP engine and the system time at which a Response occurs, if any; (ii) *throughput* – the number of triples per unit of time processed by the system; (iii) *memory usage* – the actual memory consumption of the RSP engine; (iv) *Completeness & Soundness* of query-answering results w.r.t the entailment regime that the RSP engine is exploiting.

Last, but not least, due to its case-oriented nature, SCRA exploits naive terms of comparison to draw research guidelines (Section 1). For this reason, it is necessary to identify at least one [R.6] RSP engine baseline, i.e., the minimal meaningful approaches to realize an RSP engine.

[6] `http://bit.ly/experiment-properties`

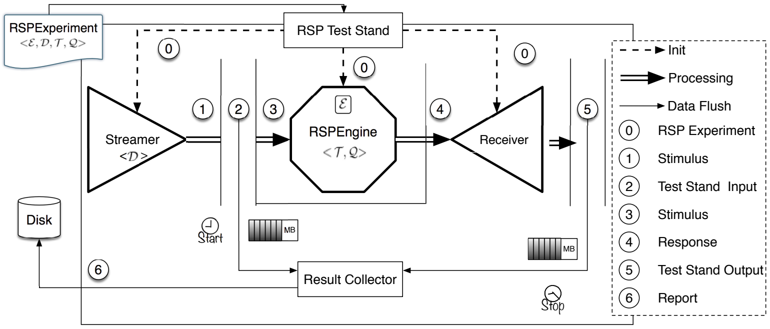


Fig. 1: RSP Test Stand modules and workflow.

4 Architecture and Workflow of RSP Test Stand

In this section, we present the architecture of a Test Stand for window based RSP engines (namely RSP Test Stand) - a software facility to enable SCRA - it components and how they interact during the execution of an RSP Experiment.

The RSP Test Stand consists of four components, whose position within the architecture is presented in Figure 1:

- The Streamer sets up the incoming streams w.r.t to $\mathcal{D}$;
- The RSP Engine represents the RSP engine $\mathcal{E}$, initialized with $\mathcal{T}$ and $\mathcal{Q}$;
- The Receiver continuously listens to $\mathcal{E}$ responses to $\mathcal{Q}$; and,
- The Result Collector compiles and persists the report $\mathcal{R}$.

During the execution, six kind of events are exchanged between the components: (i) RSP Experiment - the top-level input, defined in Section 3, used to initialize the test stand components; (ii) Stimulus - a portion of the incoming data streams in which all the data have the same application time and whose complete specification is included in $\mathcal{D}$ (e.g., a real word data stream, a recorded data stream as those proposed in [1, 11, 17, 28]), a synthetic data stream generated in order to stress the engine); (iii) Response - the answer of the queries specified in $\mathcal{Q}$; (iv) Test Stand Input - the Stimulus and the performance measurements collected as specified in $\mathcal{K}$ just before Stimulus creation; (v) Test Stand Output - the Response and the performance measurements collected as specified in $\mathcal{K}$ just after the Response creation; (vi) Test Stand Report - the carrier for the data that the Result Collector has to persist.

Figure 1 also shows also how the Test Stand components have to interact during the experiment execution. In step (0), the Test Stand receives an RSP Experiment and it initializes its modules (dashed arrows): it loads the engine $\mathcal{E}$ and registers into it $\mathcal{T}$ and the query-set $\mathcal{Q}$; it sets up the Streamer, according with the incoming streams definition $\mathcal{D}$; it connects the engine to the Receiver and it initializes the Result Collector to receives and saves the test stand outputs. The Test Stand loops between steps (1) and (4), until the experiment ends. In step (1), the Streamer creates and pushes a Stimulus to the engine $\mathcal{E}$. In step (2), the Test Stand intercepts the Stimulus of step (1), collects the performance measurements specified in $\mathcal{K}$ (e.g, it starts a timer to measure latency and it calculates the memory consumption of the system) and it sends a Test Stand Input to the Result Collector. In step (3), $\mathcal{E}$ receives a Stimulus, it processes it according with the query-set $\mathcal{Q}$ and its own execution semantics [13] (i.e., we consider the RSP engine as a black box). Step (4) occurs when $\mathcal{E}$ outputs a Response and sends it to the Receiver. In step (5), the Test Stand receives a Response from the Receiver, takes some measurements (e.g., it stops the timer and calculates the memory consumption of the system again), it creates and sends a Test Stand Output to the Result Collector. Finally, in step (6), the Result Collector persists all the collected data as an Test Stand Report.

5 Implementation Experience

In this section, we present $\mathcal{H}$eaven, an implementation of the RSP Test Stand architecture described in Section 4. In the following, we show how $\mathcal{H}$eaven satisfies the requirements we posed in Section 3.

As a consequence to requirements [R.1,R.2,R.3], we implement $\mathcal{H}$eaven with an extensible design, i.e., each module can be replaced with another one with the same interface, but different behavior, without affecting the system stability.

To satisfy [R.2] we developed a Streamer, namely RDF2RDF$_{Stream}$, that can define different types of workload. In the current implementation it allows to define (i) flows that remain stable over time; (ii) flows that remain stable for a while, then suddenly increase; and (iii) flows that change according to a distribution (e.g, Poisson or Gaussian).

The RDF2RDF$_{Stream}$ generates streams according with the $\mathcal{D}$ parameter, which comprises: DS the actual data to stream; M the data model used by the stream (e.g. RDF Streams encoded as timestamped RDF Triples or as timestamped RDF Graphs); and the function F that describes the Stimulus content over time. The RDF2RDF$_{Stream}$ (1) retrieves and parses the data from the specified file; (2) it creates a Stimulus w.r.t the specified data model M; it exploits the function F to determine the cardinality of the triple-set to stream; (3) it attaches a non-decreasing application timestamps to the Stimulus which is immediately pushed to the tested RSP engine $\mathcal{E}$. The RDF2RDF$_{Stream}$ does not consider the semantic relation between triples, but it parse the incoming data in order.

A clear limitation of $\mathcal{H}$eaven regards the satisfaction of [R.4]. Due to our choice of using Java for prototyping, we cannot explicitly control the memory consumption. A fair workaround consist in implementing $\mathcal{H}$eaven as single thread and suspending the test stand while the RSP engine is under execution. We also reduced as much as possible the number of memory-intensive data structures used in the RSP Test Stand and we limited the I/O operations.

According to [R.5], $\mathcal{H}$eaven must be independent to the KPIs set. Currently, it can measure query latency and memory consumption, but we developed it to be easily extensible with other KPIs, e.g throughput or Quality of Service etc [22].

Actually, Completeness and Soundness of the query results can be evaluated post-hoc using the Report content persisted by the Result Collector as shown in CSRBench [11].

6 Baseline RSP Engines

The final contribution of this work consists in two baseline, that implement the minimal meaningful approaches to realize an RSP engine[7] (see Section 2) and, thus, they can be used as a initial terms of comparison to enable SCRA.

They currently support reasoning under the ρDF entailment regime [19] which is the RDF-S fragment that reduces complexity while preserving its normative semantics and core functionalities. This decision is motivated because we want to provide reasoning capabilities and ρDF is the minimal meaningful task for a Stream Reasoner [26].

The baselines cover two main design decisions: (i) data can flows from the DSMS to the reasoner via snapshots (i.e. Figure 2-A) or differences (Figure 2-B); and (ii) they exploit absolute time, i.e. their internal clock can be externally controlled [9].

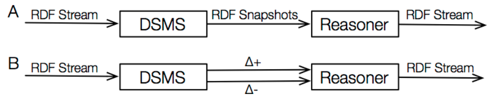


Fig. 2: The architecture of the two baselines inspired to the C-SPARQL engine (A) and of the two Incremental ones inspired to TrOWL and RDFox (B).

Figure 2-A shows the first approach, which is similar to what the C-SPARQL engine [7] does. The DSMS produces a snapshot of the active window content at each cycle and the reasoner materializes it from scratch (according to the ontology $\mathcal{T}$ and the entailment regime); then all the queries in $\mathcal{Q}$ are applied to the new materialization to generate the responses. Figure 2-B shows the second approach, that is similar to what TrOWL and RDFox [24, 20] do. The DSMS outputs the differences Δ^+ and Δ^- between the active window and the previous one. Δ^+ contains the triples that have just entered in the active window, while Δ^- contains the triples that have just exited from the active window. The reasoner, using Δ^+ and Δ^-, incrementally maintains the materialization over time.

Exploiting absolute time allows us to ensure the Completeness and Soundness of baselines responses, since we can wait for the RSP engine to complete the evaluation before sending the next Stimulus. In this way, the baselines cannot be overloaded, they can only violate the responsiveness constrain (A exhaustive reference is [9]). Moreover, we can observe the entire engine dynamics, because we fully control the engine behavior and, thus, we can relate to any given Stimulus with the relative Response even in stressing condition.

[7] We implement the baselines pipelining Esper 5.3[8], a mature open source DSMS, with the Jena 2.12.1 general purpose rule engine[9], also open source.

7 Evaluation

In this section, we demonstrate what we argued in the Section 1, i.e. (i) top-down hypothesis confirmation is not straight-forward; and (ii) $\mathcal{H}$eaven is effective because it enables relevant bottom-up analysis. To this extent, we show how to design a set of experiments setting up a controlled environment. We provide evidences of the non-obvious evaluation of hypothesis Hp.1 and Hp.2 (presented in Section 1) and we answer questions Q.a, Q.b, Q.c, Q.d for the baselines.

7.1 Experiment Design

Stimulus	ω				
Size	$.1s$	$1s$	$10s$	$100s$	$1000s$
1	1	10	10^2	10^3	10^4
10	10	10^2	10^3	10^4	
10^2	10^2	10^3	10^4		
10^3	10^3	10^4			

Table 1: The number of RDF triples in the active window as a function of the number of triples in each Stimulus and the duration ω of the window. Note sliding of the window β is 100 ms and the application time unit is 100 ms, i.e., we send a Stimulus each 100 ms.

According with RSP Experiment definition presented in Section 4, to prepare an experiment we need to fill the tuple $< \mathcal{E}, \mathcal{D}, \mathcal{T}, \mathcal{Q}, \mathcal{K} >$.

As $\mathcal{E}$, the RSP engine to test, we consider our baseline implementations.

We used RDF2RDF$_{Stream}$ as Streamer, that accepts in input any RDF file. We need to stress the reasoning capabilities of the baseline in a regular way and LUBM is a recognized benchmark for the reasoning domain that has been used before in the SR field [26][10]. Therefore, we configured $\mathcal{D}$ as follows: as dataset DS we opted for LUMB [14]; as data model M we opted for RDF Stream encoded as timestamped RDF Graphs; and as function F, since we want to keep the data stream regular, we chose one which keeps constant the number of triples in each Stimulus forming the RDF Stream and we set the application time unit of 100 ms.

Coherently with $\mathcal{E}$ and $\mathcal{D}$, we picked as $\mathcal{T}$ the ρDF subset of the LUBM TBox. As normally assumed in SR research, we consider this TBox static, therefore, the materialization of $\mathcal{T}$ is computed at configuration time.

As set of queries $\mathcal{Q}$, we choose the query that continuously asks for the entire materialization of the active window content w.r.t. $\mathcal{T}$ under ρDF entailment regime. This query is the most general query that can be registered in $\mathcal{E}$ and it is enough to support our claims. Indeed, adding more queries can only make the complex situation (without clear winners), which we discuss in Section 7.2, even more intricate. In different experiments, we used variants of the this query, fixing the sliding parameter to $\beta = 100$ ms and varying the total duration ω of the window, as summarized in Table 1.

[10] LUBM is the easiest choice in this context. Indeed, among the existing SR benchmarks only SRbench includes queries that require reasoning, but the data streams do not flow regularly. On the other hand, LUBM can be streamed regularly through the RDF2RDF$_{Stream}$ without affecting the semantics. A research towards a better SR benchmark focused on reasoning [22] is worthy, but is out of the scope of this paper.

Finally, as KPIs set $\mathcal{K}$, we measure the memory consumption and, since the baselines allows to exploit the absolute time for the computation, the query latency. To stress the baselines at maximum, we pushed the Stimuli in the baselines as soon as the baselines end the previous computation, reducing the probability of garbage collecting until the system finishes the free memory.

All experiment are 30000 Stimuli long and, each of them was executed 10 times on an 2009 iMac with 12Gb of RAM and 3.06GHZ with OSX 10.7.

7.2 Uncomfortable Truths in Hypothesis Verification

State-of-the-art results [1, 11, 17, 28] show that top-down hypothesis confirmation is not straight-forward. This is still true even under the controlled experimental condition we just defined and for already investigated hypothesis like Hp.1 and Hp.2. In the following, we specify them more w.r.t the experimental setting and we show how they are not trivially verified.

Hp.1 When $\omega = \beta$ i.e., the window contains only one Stimulus, the Naive approach is always faster than the Incremental one.

Hp.2 When the number of changes $\Delta+$ and $\Delta-$ (Section 6) is a small fraction of the content of the window an Incremental approach is faster than the Naive one.

(a) Latency

Stimulus	ω				
Size	$.1s$	$1s$	$10s$	$100s$	10^3s
1	I	I	I	I	I
10	I	I	I	I	
10^2	I	I	I		
10^3	N				

(b) Memory

Stimulus	ω				
Size	$.1s$	$1s$	$10s$	$100s$	10^3s
1	N	N	I	I	I
10	N	N	I	I	
10^2	I	I	I		
10^3	I	$\simeq$			

Table 2: Experiments comparison for average query latency (a) and average memory consumption (b) of the baselines. I:Incremental, N:Naive, $\simeq$: Even. Assumption: approach A dominates B when A/B is grater than 5%. Highlighted cells indicate where Hp.1 or Hp.2 are not confirmed.

Tables 2 (a) and (b) exploit the layout of Table 1 to compare respectively average query latency and average memory consumption values of the two approaches for all the 14 experiments. We assume that one approach dominates the other one when the differences in performance measurements are greater than the 5%.

Table 2-a seems to confirm Hp.2, but it only partially confirms Hp.1, because while the Incremental approach wins in all the settings where it was expected, the Naive one does not for two cases on four (first column).

The insights become more complex when we consider the memory consumption. Not only we have results where either Hp.1 or Hp.2 are not verified, but the results are not even coherent between memory consumption and the query latency.

Figure 3-A and Figure 3-B abstract information of Tables 2 in a dashboard representation: on the x-axis we have average latency values and on the y-axis

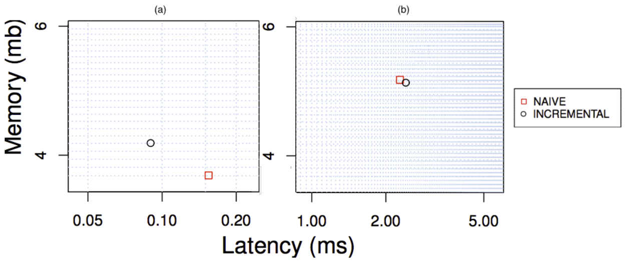


Fig. 3: Example of dashboard representation: average latency values on x-axis reports, average memory values on the y-axis in logarithmic scale. w.r.t Table 1: (a) Stimulus size = 1, ω=1s (b) Stimulus size = 1000, ω=1s

we have average memory values both in logarithmic scale. Figure 3-A shows a precise ordering between the two baselines: the Incremental is the best in term of latency while the memory consumption is quite similar. However, Figure 3-B presents an inverted situation, even though the Incremental solution has still less latency, their relative position is totally different.

We stated that any domain specific benchmark aim at fostering technological progress by guaranteeing a fair assessment, this means identify the best solution. However, this evaluation, as well as results in RSP benchmarking state-of-the-art, showed that this is not always possible: at this level of analysis, we can hardly identify in which situation a solution dominates another one and, thus, we should drill down the analysis to determine at least the situations where a solution dominates another one.

7.3 Systematic Comparative Research Approach

Answering the questions we posed in Section 1 represents a first step toward a SCRA, which does not refuse the top-down analysis, but extends it into a layered methodology that aims at catching bottom-up different aspects of the analysis.

Top down research tries to answer Q.a and Q.b applying the kind of analysis we did to verify Hp.1 and Hp.2. The top layer (dashboard) summarizes the performances of the benchmarked RSP engines into a solution space, where the engines are ordered by the experimental results. This allows to qualitatively identify the best solution (if such a dominance exists) and, thus, to answer Q.a. Intermediate layers (see Table 2) answer Q.b focusing on both *Inter Experiment Comparisons* – quantitatively contrasting the results of a single measurement in different experiments – and *Intra Experiment Comparisons* – pointing out the relation between multiple measurements within an experiment.

However, $\mathcal{H}$eaven allows also to answer Q.c and Q.d. With high level visual comparisons we can contrast many RSP engine dynamics at once and identify anomalies and/or patterns to answer Q.c. Table 3 shows how differently the memory evolves over time in different experiments just for one baseline (i.e, incremental). Finally, finer grain visual comparisons allows to contrast different variables within an experiment, and highlight the presence of internal trade-offs,

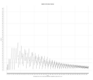
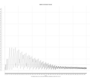
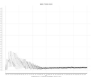
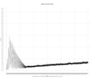
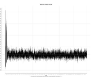
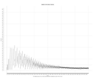
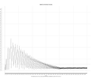
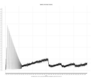
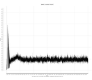
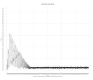
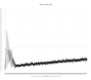
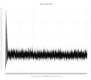
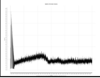
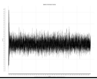
Stimulus	ω				
Size	$.1s$	$1s$	$10s$	$100s$	$1000s$
1					
10					
100					
1000					

Table 3: Pattern Identification in Memory for Incremental Baseline

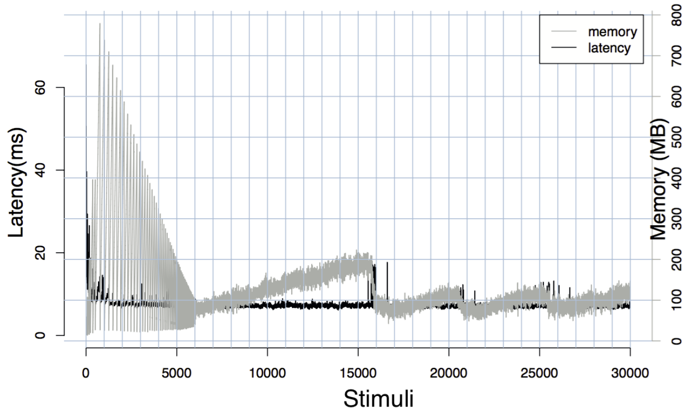


Fig. 4: Experiment 11 - Stimulus Size = 10 - ω=$1s$ - Incremental.

that may explain the surprising results we obtained previously. Indeed, Figure 4 is an example where we can observe that memory oscillations, probably due to the garbage collection, clearly influence the query latency, slowing down the system (see the latency spikes every time the memory is freed).

Enabling SCRA, by the means of $\mathcal{H}$eaven, makes possible to point out how memory and latency, as well as other KPIs, are related. For a deeper analysis of the results, we invite interested readers to check out our technical report[11].

8 Related works in RSP engines benchmarking

The state-of-the-art for RSP engine benchmarking currently comprises four benchmarks [1, 11, 17, 28], summarized in Table 4 and a first attempt towards a general architecture to systematically evaluate RSP engines [25].

LSBench [17] offers a social network data stream schema, the S2Gen generator, and 12 queries. It covers three classes of testing: i.e. a) query language

[11] http://streamreasoning.org/TR/2015/Heaven/2015-results.pdf

	Data Stream	Ontologies	Queries	Facility	Baselines
LSBench [17]	✓	✓	✓	$\simeq$	X
SRBench [28]	✓	✓	$\checkmark_{reasoning}$	X	X
CSRBench [11]	✓	✓	✓	X	$\simeq$
CityBench [1]	✓	✓	✓	$\simeq$	X
$\mathcal{H}$eaven	X	X	X	✓	✓

Table 4: RSP Benchmarking State of the art.✓: provided, X: not-provided, $\simeq$: partially-provided

expressiveness; b) maximum execution throughput and scalability in terms of query number and static data size; c) result mismatch between different engines to assess the correctness of the answers.

SRBench [28] focuses on RSP engine query language coverage. It comprises 17 queries to cover three main use cases: (i) flow-data only (ii) flow and background data and (iii) flow and GeoNames and DBpedia datasets. It also targets the reasoning task testing, but not as a main use case. CSRBench [11] extends [28] by addressing the correctness verification. It adds to the query set aggregated queries, queries requiring to join triples with different timestamp and parametric sliding windows queries. Moreover, it provides an Oracle to automatically check correctness of query results.

CityBench [1] provides a real world data streams from the CityPulse project[12] (i.e., vehicle traffic data, weather data and parking spots data); a synthetic data streams about user location and air pollution. Real world static datasets about cultural events are also included. [1] evaluates query latency, memory consumption and result completeness as evaluation metrics. 13 continuous queries are available to test the RSP engine. It also provides a testbed infrastructure for experiment execution, that specifically targets smart city workloads and applications.

Table 4 shows the positioning of this work w.r.t [1, 11, 17, 28]. Differently from the state-of-the-art, we do not propose new workloads, queries and ontologies, but we target the evaluation approach itself. Our aim is to enable SCRA for window based RSP engines rather than proposing yet another benchmark.

Moreover, it is worth to note that this work differs from the one we presented in [25], because in this paper we provides a broader set of requirements, we revised the architecture, we re-implement the proof of concept of the architecture; we also introduce two baselines to be used as terms of comparison and we evaluate the effectiveness of $\mathcal{H}$eaven using them.

9 Conclusions

In this paper, we claimed that the RSP community needs to enable a Systematic Comparative Research Approach for window-based RSP engines. To support this claim and enable SCRA, we proposed: (i) set of requirements elicited on the experiment definition; (ii) a general architecture for an RSP engine Test Stand [25]; (iii) $\mathcal{H}$eaven, a proof-of-concept of such architecture that is released

[12] http://citypulse.insight-centre.org/

as open source; and (iv) two *baselines*, i.e, the minimal meaningful approaches to realize an RSP engine, because the SR community still miss well defined terms of comparison that other communities have (e.g, machine learning algorithms).

We demonstrate the framework effectiveness running a set of experiments that involve $\mathcal{H}$eaven and the baselines.

We successfully showed that top-down hypothesis verification is not straight forward by proving that, even when an RSP engines is extremely simple (i.e. the baselines), it is hard to formulate verifiable hypothesis. We also proved $\mathcal{H}$eaven effectiveness by showing how it reduces the investigation biases, allowing to better understand ambiguous results, by enabling to catch cross-layered insights.

Finally, we positioned this work in the state of the art of RSP benchmarking, showing its novelty and the differences with similar solution [1, 25].

This work is towards an RSP Engine lab, an web-based environment where a users can: (i) design an experiment, (ii) run it, (iii) visualize and (v) compare the results. Therefore, we have to: (i) implement the adapting facade for the RSP engines to test and (ii) include, in a experiments suite, existing workload, ontologies and queries (e.g those of SRBench and LSBench and CityBench). Finally the baselines will shepherd future RSP comparative researches, due to the simplicity of their architectures, the availability of full modeled execution semantics [12, 21] and their open source implementation that allows to fully observe their dynamics. Therefore, we want to pursuit the baselines development.

References

1. Ali, M.I., Gao, F., Mileo, A.: Citybench: A configurable benchmark to evaluate rsp engines using smart city datasets. In: The Semantic Web-ISWC 2015. pp. 374–389. Springer (2015)
2. Anicic, D., Fodor, P., Rudolph, S., Stojanovic, N.: EP-SPARQL: a unified language for event processing and stream reasoning. In: WWW. pp. 635–644 (2011)
3. Babcock, B., Babu, S., Datar, M., Motwani, R., Widom, J.: Models and issues in data stream systems. In: SIGMOD. pp. 1–16 (2002)
4. Balduini, M., Della Valle, E., Dell'Aglio, D., Tsytsarau, M., Palpanas, T., Confalonieri, C.: Social listening of city scale events using the streaming linked data framework. In: ISWC Part II. pp. 1–16 (2013)
5. Barbieri, D.F., Braga, D., Ceri, S., Della Valle, E., Grossniklaus, M.: C-SPARQL: A continuous query language for RDF data streams. IJSC 4(1), 3–25 (2010)
6. Barbieri, D.F., Braga, D., Ceri, S., Della Valle, E., Grossniklaus, M.: Incremental reasoning on streams and rich background knowledge. In: ESWC 2010, Part I. pp. 1–15 (2010)
7. Barbieri, D.F., Braga, D., Ceri, S., Della Valle, E., Grossniklaus, M.: Querying RDF streams with C-SPARQL. SIGMOD Record 39(1), 20–26 (2010)
8. Calbimonte, J.P., Corcho, O., Gray, A.J.G.: Enabling ontology-based access to streaming data sources. In: ISWC. pp. 96–111 (2010)
9. Cugola, G., Margara, A.: Processing flows of information: From data stream to complex event processing. ACM Comput. Surv. 44(3), 15:1–15:62 (Jun 2012)
10. Della Valle, E., Ceri, S., Barbieri, D.F., Braga, D., Campi, A.: A first step towards stream reasoning. In: FIS. pp. 72–81 (2008)

11. Dell'Aglio, D., Calbimonte, J., Balduini, M., Corcho, Ó., Della Valle, E.: On correctness in RDF stream processor benchmarking. In: ISWC. pp. 326–342 (2013)
12. Dell'Aglio, D., Della Valle, E.: Incremental reasoning on rdf streams. In: Harth, A., Hose, K., Schenkel, R. (eds.) Linked Data Management, chap. 16, pp. 413–436. CRC Press (2014)
13. Dell'Aglio, D., Della Valle, E., Calbimonte, J.P., Corcho, O.: Rsp-ql semantics: A unifying query model to explain heterogeneity of rdf stream processing systems. International Journal on Semantic Web and Information Systems (IJSWIS) 10(4), 17–44 (2014)
14. Guo, Y., Pan, Z., Heflin, J.: LUBM: A benchmark for OWL knowledge base systems. Journal of Web Semantics 3(2-3), 158–182 (2005)
15. Kuehl, R.O.: Design Of Experiments Stastistical Principles Of Research Design And Analysis. No. Q182. K84 2000. (2000)
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: ISWC. pp. 370–388 (2011)
17. Le-Phuoc, D., Dao-Tran, M., Pham, M.D., Boncz, P., Eiter, T., Fink, M.: Linked stream data processing engines: Facts and figures. In: ISWC. pp. 300–312 (2012)
18. Margara, A., Urbani, J., van Harmelen, F., Bal, H.E.: Streaming the web: Reasoning over dynamic data. J. Web Sem. 25, 24–44 (2014)
19. Muñoz, S., Pérez, J., Gutiérrez, C.: Minimal deductive systems for RDF. In: ESWC. pp. 53–67 (2007)
20. Nenov, Y., Piro, R., Motik, B., Horrocks, I., Wu, Z., Banerjee, J.: Rdfox: A highly-scalable rdf store. In: The Semantic Web-ISWC 2015, pp. 3–20. Springer (2015)
21. Ren, Y., Pan, J.Z.: Optimising ontology stream reasoning with truth maintenance system. In: CIKM. pp. 831–836 (2011)
22. Scharrenbach, T., Urbani, J., Margara, A., Della Valle, E., Bernstein, A.: Seven commandments for benchmarking semantic flow processing systems. In: ESWC. pp. 305–319 (2013)
23. Tatbul, N., Cetintemel, U., Zdonik, S., Cherniak, M., Stonebraker, M.: Exploiting punctuation semantics in continuous data streams. IEEE Trans. on Knowledge and Data Eng. 15(3), 555–568 (2003)
24. Thomas, E., Pan, J.Z., Ren, Y.: Trowl: Tractable OWL 2 reasoning infrastructure. In: ESWC Part II. pp. 431–435 (2010)
25. Tommasini, R., Della Valle, E., Balduini, M., Dell'Aglio, D.: Heaven test stand: towards comparative research on rsp engines. OrdRing (2015)
26. Urbani, J., Margara, A., Jacobs, C.J.H., van Harmelen, F., Bal, H.E.: Dynamite: Parallel materialization of dynamic RDF data. In: ISWC Part I. pp. 657–672 (2013)
27. Walavalkar, O., Joshi, A., Finin, T., Yesha, Y.: Streaming knowledge bases. In: In International Workshop on Scalable Semantic Web Knowledge Base Systems (2008)
28. Zhang, Y., Duc, P., Corcho, O., Calbimonte, J.P.: SRBench: A Streaming RDF/SPARQL Benchmark. In: ISWC. pp. 641–657 (2012)