RSPLab: RDF Stream Processing Benchmarking Made Easy

Riccardo Tommasini, Emanuele Della Valle, Andrea Mauri, Marco Brambilla

Politecnico di Milano, DEIB, Milan, Italy
`{name.lastname}@polimi.it`

Abstract. In Stream Reasoning (SR), empirical research on RDF Stream Processing (RSP) is attracting a growing attention. The SR community proposed methodologies and benchmarks to investigate the RSP solution space and improve existing approaches. In this paper, we present RSPLab, an infrastructure that reduces the effort required to design and execute reproducible experiments as well as share their results. RSPLab integrates two existing RSP benchmarks (LSBench and CityBench) and two RSP engines (C-SPARQL engine and CQELS). It provides a programmatic environment to: deploy in the cloud RDF Streams and RSP engines, interact with them using TripleWave and RSP Services, and continuously monitor their performances and collect statistics. RSPLab is released as open-source under an Apache 2.0 license.

Keywords: Semantic Web, Stream Reasoning, RDF Stream Processing, Benchmarking

1 Introduction

In the recent years, research about Semantic Web and streaming data – Stream Reasoning (SR) – constantly grew. The community has been investigating foundational research on algorithms for RDF Stream Processing (RSP) [6], applied research with systems architectures [3, 10] and, recently, empirical research on benchmarks [15, 11, 5, 1, 8] and evaluation methodologies [14, 17, 12].

Focusing on the latter two, the state of the art comprehends RSP engines prototypes [3, 10] and benchmarks that address the different challenges the community investigated: query language expressive power [15], performance [11], correctness of results [5, 8], memory load and latency [1, 8]. This heterogeneity of benchmarks helps to explore the solution space, but hinders the systematic evaluation of RSP engines. Therefore, [14] proposed a requirement analysis for benchmarks and ranked existing benchmark accordingly; [17] proposed a framework for systematic and comparative RSP research. Beside the aforementioned community efforts, the evaluation of RSP engines is still not systematic.

In this paper, we propose RSPLab [18] a cloud-ready open-source test driver to support empirical research for SR/RSP. RSPLab offers a programmatic environment design and execute experiments. It uses linked data principle to publish RDF streams [9] and a set of REST APIs [2] to interact with RSP engines.

RSPLab continuously monitors memory consumption and CPU load of the deployed RSP engines and it persists the measurements on a time-series database.

It allows to estimate results correctness and max throughput post-hoc by collecting query results on a reliable file storage. RSPLab provides real-time assisted data visualization by the means of a dashboard. Finally, it allows to publish experimental reports as linked data.

2 RSPLab

In this section, we present the requirements for a RSP test driver, we describe the test driver architecture and how RSPLab currently implements it.

Requirements. We elicit the requirements for a test driver considering the existing research on benchmarking of RSP systems. We focused on the different engines involved, the data used and the applied methodologies. Therefore, our requirements analysis comprises:

(R.1) *Benchmarks Independence.* RSPLab must allow its users to integrate any benchmark, i.e. ontologies, streams, dataset and queries.
(R.2) *Engine Independence.* RSPLab must be agnostic to the RSP engine under test and it must not be bounded to any specific query language (QL).
(R.3) *Minimal yet Extensible KPI set.* According to the state of the art [1, 14, 17], the KPI set must include at least query result correctness and throughput. However, the KPI set must be extensible to include KPIs that are measurable in specific implementation and deployment.
(R.4) *Continuous Monitoring.* RSPLab must enable the observation of the RSP engine dynamics under the whole experiment execution.
(R.5) *Error Minimization.* RSPLab must minimize the experimental error, isolating each module to avoid resource contention.
(R.6) *Ease of Deployment.* RSPLab must be easy-to-deploy and it must simplify the deployment of the experiments modules, e.g. streams and engines.
(R.7) *Ease of Execution.* RSPLab must simplify the access to the available resource, e.g. reuse existing benchmarks, and the execution of experiments.
(R.8) *Repeatability* RSPLab must guarantee experiment repeatability under the specific settings.
(R.9) *Data Analysis.* RSPLab must render simple data analyses about the collected statistics and allow its users to perform custom ones.
(R.10) *Data Publishing.* RSPLab must simplify the publications of performance statistics, query results and experiment design using linked data principles.

Architecture. Figure 1 presents RSPLab architecture that comprises four independent tiers: *Streamer*, *Consumer*, *Collector* and *Controller*. For each tier, it shows its logical submodules, e.g., a timeseries database in the Collector, and it refers to the technologies involved in the current implementation, e.g. InfluxDB.

The *Streamer*, the data provisioning tier, publishes RDF streams from existing benchmarks **(R.1)**. The *Streamer* can stream any (virtual) RDF dataset that has a temporal dimension. Published RDF streams are accessible from the web. The *Consumer*, the data processing tier, exposes the RSP engines on the web by the mean of REST APIs **(R.2)**. The minimal required method comprise source, query and sinks registration **(R.1)**.

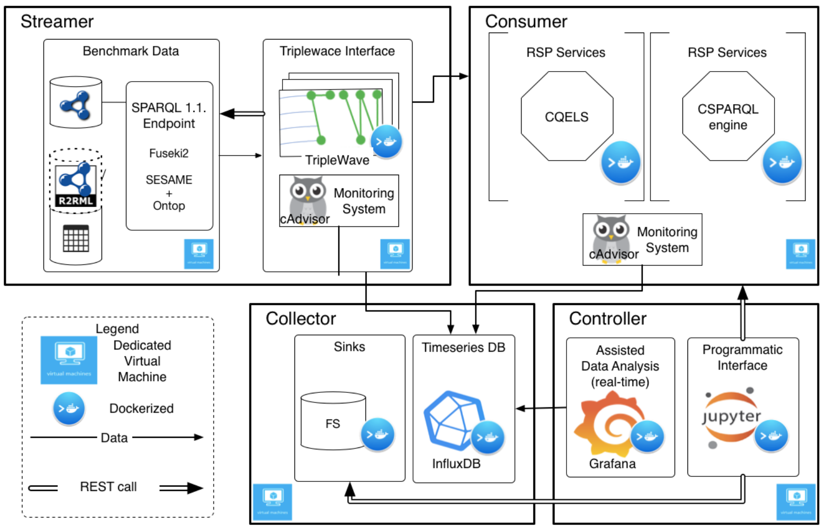

Fig. 1: RSPLab Architecture and Implementation

The *Collector*, the monitoring tier, comprises two submodules: (1) a *monitoring system* that, during the executions of experiments, continuously measures the performance statistics of any deployed module **(R.4)**, (2) a time-series database to save the statistics and a persistent storage to save the query results. **(R.3)**. The *Controller*, the control and analysis tier, allows the RSPLab user to control the other tiers. It allows to design and execute the experiments programmatically **(R.7)**. It enable the verification of the results **(R.8)**. through an assisted and customized real time data analysis dashboard **(R.9)**.

Implementation Experience. To develop RSPLab, we used Docker, i.e. a lightweight virtualization framework[1]. Docker simplifies the deployment process, it reduces the biases and foster the reproducibility of experiments [4]. As any virtualization theniques, it grants full control to the available resources, allowing to scale the virtual infrastructure **(R.6)**. It minimizes the experimental error **(R.5)** by guaranteeing components isolation. Moreover, it fosters reproducibility by making the execution hardware-independent **(R.8)**. Figure 1 illustrates how we deployed RSPLab's components in independent virtual machines. It also shows how the dockerization is done and it references the used technologies. RSPLab is natively deployable on AWS[2] and Azure[3] infrastructures **(R.6)**.

Streamer. This tier is implemented using a modified version of TripleWave[9][4,5] that includes methods to registers and start streams remotely. It includes syn-

[1] `https://www.docker.com/`
[2] `https://aws.amazon.com`
[3] `https://azure.microsoft.com`
[4] `https://github.com/streamreasoning/triplewave/tree/rsplab`
[5] `https://hub.docker.co/r/streamreasoning/triplewave/`.

thetic RDF data from LSBench. We used the included data generator and we loaded them into a SPARQL endpoint to stream with TripleWave. It also includes data from CityBench. We exploited R2RML mappings and to convert CSV data into RDF on-demand. This tier is not limited to them. Streams from other benchmarks can be added following TripleWave principles.
Consumer. This tier uses the RSP Services [2], i.e. a set of REST methods that abstract from the RSP engine's query language syntax and semantics. The RSP services generalize the processing model enabling streams registration, queries registration and results consumption. This tier includes, but it not limited to, CQELS [10] and C-SPARQL [3] engine. Using the RSP Services, new RSP engines can be added to RSPLab.
Collector. This tiers includes (1) a distributed continuous monitoring system, called cAdvisor[6], that collects statistics about memory consumption, CPU load every $100ms$ **(R.3)** for Docker containers. We target those running RSP engines but any of RSPLab's component can be observed. (2) A time-series database, called InfluxDB,[7] where we write the collected statistics. (3) A python daemon, called RSPSink, that persists query results on a cloud file systems (e.g., Amazon S3 or Azure Blob Storage), allowing to verify correctness and estimate the system's maximum throughput post-hoc.
Controller. This tier is implemented using iPython Notebooks[8]. We developed an ad-hoc python library [16] that allows to interact with the whole environment. It includes wrappers to RSP services, TripleWave APIs and sinks. Thanks to this programmatic APIs the RSPLab user can run TripleWave and RSP engine instances, execute experiment over them and analyze the results in a programmatic way **(R.7)**. Moreover, with Grafana[9] it provides an assisted data visualization dashboard that reads data from InfluxDB enabling real-time monitoring **(R.9)**. Last, but not least, the included library automatically generates experiments reports using the VOID vocabulary **(R.10)**.

3 RSPLab In-Use

In this section, we show how to design and execute experiments and how to publish the results as linked using RSPLab.

	Citybench
$\mathcal{E}$	CSPARQL engine
$\mathcal{K}$	Memory & CPU
$\mathcal{T}$	citybench ontology
$\mathcal{D}$	SensorRepository
$\mathcal{S}$	Aarhus Traffic Data 182955 & 158505
$\mathcal{Q}$	Q1

Table 1: The running experiment.

Experiment Design. For this process, we consider the following experiment definition from [17]. An RSP experiment is a tuple $\langle \mathcal{E}, \mathcal{K}, \mathcal{Q}, \mathcal{S}, \mathcal{T}, \mathcal{D} \rangle$ where $\mathcal{E}$ is the RSP engine subject of the evaluation, $\mathcal{K}$ is the set of KPIs measured in the experiment, i.e., those included in RSPLab. $\mathcal{Q}$ is the set of continuous queries that the

[6] `https://github.com/google/cadvisor`
[7] `https://www.influxdata.com/`
[8] `https://ipython.org/notebook.html`
[9] `https://grafana.com/`

```
_Q1 = rsp.BenchmarkQueries.CityBench.Q1
e = rsp.new_experiment()
e.add_engine("http://cqles.rsp-lab.org", 80, rsp.Dialects.CQELS)
e.add_KPIs(rsp.KPI.Memory_Consumption, rsp.KPI.CPU_Load)
e.add_query("CB.Q1", rsp.QueryType.Query, _Q1, rsp.Dialects.CQELS)
e.add_tbox("CB.Q1", name="citytraffic.owl", base="rsp-lab.org")
e.add_graph("CB.Q1", name="SensorRepository.rdf", base="rsp-lab.org")
e.add_stream("CB.Q1","AarhusTrafficData158505", base="rsp-lab.org")
e.add_stream("CB.Q1","AarhusTrafficData182955", base="rsp-lab.org")
```

Listing 1.1: RSP Experiment Design RSPLab

engine has to answer. $\mathcal{S}$ is the set of RDF streams required to the queries in $\mathcal{Q}$. Finally, $\mathcal{T}$, $\mathcal{D}$ are, respectively, the static set of terminological axioms (TBox), and the static RDF datasets.

Table 1 shows an example of an experiment that can be defined within RSPLab. We took this example from the Citybench benchmark. The engine used is C-SPARQL, the observed measures are Memory and CPU load, the TBox is the *citybench* ontology and the RDF dataset involved is *SensorRepository*. The query-set consists of the only query *Q1* which utilizes data coming from two traffic streams (e.g., *AarhusTrafficData182955* and *AarhusTrafficData158505*). Listing 1.1 shows how to create the experiment with the included python library. All the static data, streams and queries are available at on GitHub[10].

```
#WARM-UP
rsp = RSPEngine(ehost, eport);

for d in experiment.graphs():
  rsp.register_graph(d)
for s in experiment.streams():
  rsp.register_stream(s)
for q in experiment.queries():
  rsp.register_query(q)
  rsp.new_observer(query,'
      default')
spawn_sinks(experiment)

# OBSERVE
wait(experiment.duration())

for q in engine.queries():
   for o in engine.observers(q):
      rsp.unregister_observer(o)
   rsp.unregister_query(q)
for s in engine.streams():
   rsp.unregister_stream(s)
rsp.report.publish(experiment)
```

Listing 1.2: Execution with RSPLab.

```
@prefix:<http://rsp-lab.org/vocab/>

:RSPLab a foaf:Organization;
:exp1 a void:Dataset;
 dcterms:subject <:CSPARQL_Engine>;
 dcterms:contributor :RSPLab;
 dcterms:created "2017/05/08";
 dcterms:license <http://..cc/>;
 void:subset :exp, :results, :cpu .

:exp a void:Dataset;
 void:dataDump <exp1.jsonld>
 dcterms:subject <:Experiment> .

:cpu a void:Dataset;
 void:dataDump <exp1-cpu.csv>;
 dcterms:subject <rsplab:CPULoad> .

:results a void:Dataset;
  void:subset [
    dcterms:subject <:Results>;
    void:feature <format:Turtle>;
    void:dataDump <Q1.ttl> .]
```

Listing 1.3: Experimental Report with VoID.

Experiment Execution. In RSP, the experimental workflow has a warm-up phase followed by an observation phase because most of the transient behaviors occur during the engine warm-up and they should not bias the performance measures [12, 1, 11].

Warm-Up. In this phase, RSPLab deploys engine and RDF streams. It registers the streams, the queries and the observers on the RSP engine subject of the evaluation. It sets up the sinks to persists the queries results. Observing

[10] https://github.com/streamreasoning/rsplab, at rsplab/streamer/citybench/setup/

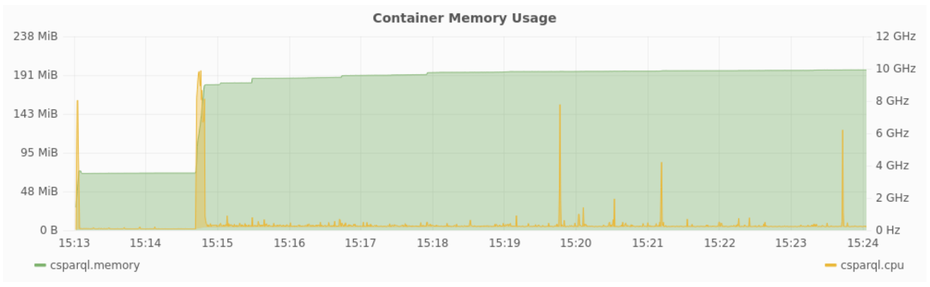

Fig. 2: C-SPARQL engine CPU and Memory usage.

the engine's dynamics using the assisted dashboard (Grafana) and it possible to determinate when the RSP engine is steady. Listings 1.2, lines 1 to 14, shows how this phase looks like in RSPLab. Figure 2 shows how this phase impacts the system dynamics approximatively until 15.16.

OBSERVE. In this phase, which usually has a fixed duration, the RSP engine is stable. It consumes the streams and answers the queries. The results and the performance statistics are persisted. When time expires, everything is shut down. Listings 1.2, lines 15 to 24 shows how this phase looks like in RSPLab. RSPLab makes possible to define more complex workflows, simulate real scenarios, e.g. add/remove queries or tune stream rates while observing the engine response.

Report & Analysis. RSPLab automatically collects performance statistics and enable experiment reporting using linked data principles. An example of data visualization using the integrated dashboard is in Figure 2. Listings 1.3 shows an example of experimental done with RSPLab that uses VOID vocabulary to publish experiment design, CPU performance metrics and query results.

4 Related Work

In this section, we compare RSPLab with existing research solutions from SR/RSP, Linked Data, and database.

LSBench's and Citybench [11, 1] proposed two test-drivers that push RDF Stream to the RSP engine subject of the evaluation. Differently from RSPLab, they are not benchmark-independent **(R.1)**. The test drivers are designed to work with the benchmark queries and stream the benchmark data and do not guarantee error minimization by the means of module isolation**(R.5)**.

Heaven [17] includes a test-bed proof of concept, with an architecture similar to RSPLab. However, Heaven does not include a programmatic environment that simplifies experiment execution **(R.7)**, is not engine-independent **(R.2)**, and its scope is limited to window-based, single-thread RSP engines. Like RSPLab, Heaven treats RSP engines as black box, but communication happens using Java Facade rather than a RESTful interface. Therefore, Heaven constrains the RSP engine's processing model. It enables analysis of performance dynamics but it does not offer assisted data visualization **(R.9)** nor automated reporting **(R.10)**.

LOD Lab [13] aims at reducing the human cost of approach evaluation. It also supports data cleaning and simplifies dataset selections using metadata. However, RDF Streams and RSP engine testing are not in its scope. LOD Lab does not offer a continuous monitoring system, but only addresses the problem of data provisioning. It provides a command line interface to interact with it **(R.6)**, but not a programmatic environment to control the experimental workflow **(R.7)**.

OLTP-Bench [7] is a universal benchmarking infrastructure for relational databases. Similarly to RSPLab it supports the deployment in a distributed environment **(R.6)** and it comes with assisted statistics visualization **(R.9)**. However, it does not offer a programmatic environment to interact with the platform, execute experiments **(R.7)** and publish reports **(R.10)**. OLTP-Bench includes a workload manager, but does not consider RDF Streams. Moreover, it provides an SQL dialect translation module, which is flexible enough in the SQL area but not in the SR/RSP one **(R.2)**.

5 Conclusion

This paper presented RSPLab, a test-drive for SR/RSP engines that can be deployed on the cloud. RSPLab integrates two existing RSP benchmarks (LSBench and CityBench) and two exiting RSP engines (C-SPARQL engine and CQELS). We showed that it enables design of experiments by the means of a programmatic interface that allows deploying the environment, running experiments, measuring the performance, visualizing the results as reports, and cleaning up the environment to get ready for a new experiment.

RSPLab is released as open-source citable[18, 16] and available at `rsp-lab.org` Examples, documentation and deployment guides are available on GitHub hosted by the Stream Reasoning organization.

Future work on RSPLab comprise (i) the integration of all the existing RSP benchmarks datasets and queries, i.e. SRBench and YaBench, (ii) the integration of CSRBench's and YABench's oracles for correctness checking (iii) the execution of existing benchmark experiments at scale and systematically. Last, but not least, (iv) the extension of RSPLab APIs towards a RSP Library.

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 - 14th ISWC, Bethlehem, PA, USA, October 11-15, 2015, Proceedings, Part II. pp. 374–389
2. Balduini, M., Valle, E.D.: A restful interface for RDF stream processors. In: Proceedings of the ISWC 2013 Posters & Demonstrations Track, Sydney, Australia, October 23, 2013. pp. 209–212
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. Semantic Computing 4(1), 3–25
4. Boettiger, C.: An introduction to docker for reproducible research. Operating Systems Review 49(1), 71–79 (2015), `http://doi.acm.org/10.1145/2723872.2723882`

5. Dell'Aglio, D., Calbimonte, J., Balduini, M., Corcho, Ó., Della Valle, E.: On correctness in RDF stream processor benchmarking. In: The Semantic Web - ISWC 2013 - 12th ISWC, Sydney, NSW, Australia, October 21-25, 2013, Proceedings, Part II. pp. 326–342
6. Dell'Aglio, D., Della Valle, E., Calbimonte, J., Corcho, Ó.: RSP-QL semantics: A unifying query model to explain heterogeneity of RDF stream processing systems. Int. J. Semantic Web Inf. Syst. 10(4), 17–44
7. Difallah, D.E., Pavlo, A., Curino, C., Cudré-Mauroux, P.: Oltp-bench: An extensible testbed for benchmarking relational databases. PVLDB 7(4), 277–288
8. Kolchin, M., Wetz, P., Kiesling, E., Tjoa, A.M.: Yabench: A comprehensive framework for RDF stream processor correctness and performance assessment. In: Web Engineering - 16th International Conference, ICWE 2016, Lugano, Switzerland, June 6-9, 2016. Proceedings. pp. 280–298
9. Mauri, A., Calbimonte, J., Dell'Aglio, D., Balduini, M., Brambilla, M., Valle, E.D., Aberer, K.: Triplewave: Spreading RDF streams on the web. In: The Semantic Web - ISWC 2016 - 15th International Semantic Web Conference, Kobe, Japan, October 17-21, 2016, Proceedings, Part II. pp. 140–149
10. Phuoc, D.L., Dao-Tran, M., Parreira, J.X., Hauswirth, M.: A native and adaptive approach for unified processing of linked streams and linked data. In: The Semantic Web - ISWC 2011 - 10th International Semantic Web Conference, Bonn, Germany, October 23-27, 2011, Proceedings, Part I. pp. 370–388
11. Phuoc, D.L., Dao-Tran, M., Pham, M., Boncz, P.A., Eiter, T., Fink, M.: Linked stream data processing engines: Facts and figures. In: The Semantic Web - ISWC 2012 - 11th ISWC, Boston, MA, USA, November 11-15, 2012, Proceedings, Part II. pp. 300–312
12. Ren, X., Khrouf, H., Kazi-Aoul, Z., Chabchoub, Y., Curé, O.: On measuring performances of C-SPARQL and CQELS. CoRR abs/1611.08269
13. Rietveld, L., Beek, W., Schlobach, S.: LOD lab: Experiments at LOD scale. In: The Semantic Web - ISWC 2015 - 14th International Semantic Web Conference, Bethlehem, PA, USA, October 11-15, 2015, Proceedings, Part II. pp. 339–355 (2015), `http://dx.doi.org/10.1007/978-3-319-25010-6_23`
14. Scharrenbach, T., Urbani, J., Margara, A., Della Valle, E., Bernstein, A.: Seven commandments for benchmarking semantic flow processing systems. In: The Semantic Web: Semantics and Big Data, 10th International Conference, ESWC 2013, Montpellier, France, May 26-30, 2013. Proceedings. pp. 305–319
15. Stupar, A., Michel, S.: Srbench-a benchmark for soundtrack recommendation systems. In: 22nd ACM International Conference on Information and Knowledge Management, CIKM'13, San Francisco, CA, USA, October 27 - November 1, 2013. pp. 2285–2290
16. Tommasini, R.: streamreasoning/rsplib: Rsplib beta v0.2.4 (May 2017), `https://doi.org/10.5281/zenodo.579659`
17. Tommasini, R., Della Valle, E., Balduini, M., Dell'Aglio, D.: Heaven: a framework for systematic comparative research approach for rsp engines. In: 13th Extended Semantic Web Conference, ESWC 2016, Heraklion, Crete, Greece. pp. 87–92
18. Tommasini, R., Mauri, A.: streamreasoning/rsplab: Rsplab v0.9 (May 2017), `https://doi.org/10.5281/zenodo.572320`