# WAVES: Big Data Platform for Real-time RDF Stream Processing

Houda Khrouf, Badre Belabbess, Laurent Bihanic, Gabriel Képéklian, Olivier Curé

# WAVES: Big Data Platform for Real-time RDF Stream Processing

Houda Khrouf[1], Badre Belabbess[1,2], Laurent Bihanic[1], Gabriel Kepeklian[1] and Olivier Curé[2]

[1] ATOS, F-95870, Bezons, France. {firstname.lastname}@atos.net
[2] LIGM (UMR 8049), F-77454, Marne-la-Valle, France.
{firstname.lastname}@u-pem.fr

**Abstract.** Processing data as they arrive has recently gained momentum to mine continuous, high-volume and unbounded sequence of data streams. Due to the heterogeneity and the multi-modality of this data, RDF is widely used to provide a unified metadata layer in streaming context. In response to this ever-increasing demand, a number of systems and languages were produced, aiming at RDF stream processing (RSP). However, most of them adopt a centralized execution approach which puts a barrier to ensure correct behavior and high scalability under certain circumstances such as concurrent queries and increasing input load. Only few systems sought to distribute processing, but their implementation is still in its infancy. None of them provide a full-fledged and production-ready RSP engine that is easy-to-use, supports all SPARQL 1.1 operators and adapted to industrial needs. As a solution, we present a distributed, fault-tolerant and scalable RSP system that exploits the Apache Storm framework.

## 1 Introduction

In most parts of the world, fast growing urbanization has faced several challenges throughout the last decades. Achieving sustainable, environment-friendly and highly operating cities for a better quality of life requires innovative solutions brought by cutting edge technology. Creative ways of thinking have opened a brand new world of possibilities such as smart grids, smarter buildings and smart transportation systems. Most of these systems exploit sensor networks, an important component of the Internet Of Things, forcing to provide spatio-temporal processing that shares characteristics with Big Data ecosystems. Indeed processing streaming data requires technologies to face the high volume and velocity at which this data is coming into the system. Moreover, due to the heterogeneous nature of streaming data, the Semantic Sensor Web [11] has been designed with the aim to increase interoperability and derive contextual information. As such, processing semantic sensor streams has been the focus of different RSP engines. Well-known examples are C-SPARQL [3] and CQELS [8] which provide their own architecture, SPARQL-like query language and operational semantics. They make use of continuous queries based on windowing techniques as defined

in Data Stream Management Systems (DSMSs), and they enable reasoning over dynamic data. However, none of them could handle massive streaming data and address scalability issues as they have been mainly designed to run on a single machine. A recent RSP benchmark [7] evaluating the above systems highlights the decrease of precision and recall of output results when increasing input load.

To address the scalability issues, efforts have been made to design a new generation of RSP engines based on modern big data frameworks. Their implementation is still at an early stage and their usage is not practical when it comes to execute simple SPARQL queries. To remedy these limitations, we propose the so-called WAVES[3], a new distributed RSP system built on top of popular big data frameworks and supporting continuous queries over event-based streaming data. In a nutshell, sensor data which could be sampled are transformed into an RDF format, compressed and sent to a messaging broker. Then, a distributed system is invoked to handle windowing operations, querying and rendering results. The system is open-source[4], fully configurable, easy-to-use and general-purpose.

The remainder of this paper is organized as follows. In Section 2, we provide an overview of related work. Then, we describe the architecture of WAVES in Section 3, and we detail our data compression strategy in Section 4. Section 5 presents a preliminary evaluation, and Section 6 provides the conclusion with an outlook on future work.

## 2 Related Work

In the past few years, dynamic data streams have attracted considerable interest within the semantic web community. Processing these streams has recently been the focus of RSP systems which are mostly based on centralized execution. Recognizing the scalabability limitations of single-machine systems, efforts have relied on generic stream processing frameworks such as S4 [9] and Storm [12] to distribute querying over a cluster of machines. For example, CQELS-Cloud [10] is the first distributed system which focuses on elasticity and scalability aspects such as memory size, network bandwidth and process speed. The architecture of CQELS Cloud consists of one Execution Coordinator and several Operator Containers. The Execution Coordinator is used for distributing processing tasks and each Operator Container is used to execute one single operation such as union, aggregation, etc. Although the approach seems interesting, the system is not generic enough and does not allow end users to define customized queries. Indeed, to the best of our knowledge, CQELS Cloud is not open source which obstructs its application in many use cases. Moreover, according to the link[5] provided by CQELS team, users need to modify the source code in order to define their own queries and input data. They need also to modify several parameters (e.g., number of executors for each task, type of aggregation, number of output buffers) which is not practical for industrial applications. As we cannot access

[3] Details on WAVES project available at `http://waves-rsp.org/`
[4] It will be available on GitHub. Please contact us at: `contact[at]waves-rsp.org`
[5] `https://code.google.com/archive/p/cqels/wikis/CQELSCloud.wiki`

the source code or run our queries, we did not consider this system in our experiments. Another RSP engine called KATTS [5] has been proposed based on graph partitioning algorithm to optimize the network bandwidth and distribute querying. However, it does not support the SPARQL syntax and users are obliged to transform a query into a XML tree format. Besides, several SPARQL operators are not supported such as `OPTIONAL`, `LIMIT`, `SUM` to name a few. Querying over static data is also not possible. Different from these systems, our design aims to provide a generic, complete and production-ready distributed RSP engine that could be easily used and practical for various applications.

## 3 WAVES Architecture

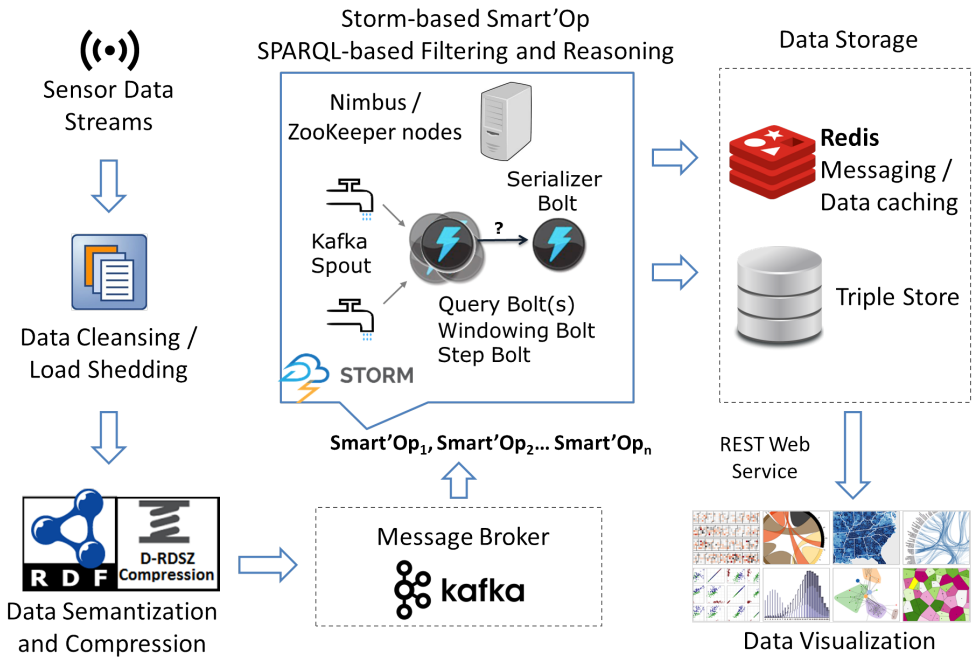



**Fig. 1:** WAVES Architecture overview

Figure 1 depicts an overview of WAVES architecture. The system processes dynamic and event-based data which are handled by distributed components, i.e., Redis [2] as an in-memory store, Kafka [13] as a publish-subscribe distributed message broker and Storm [12] as a distributed stream processing framework. A typical scenario in our system is as follows. First, measures are captured from

a given sensor network. Events from these streams are cleansed, filtered and possibly sampled before being transformed into a compressed RDF format and submitted to the Kafka message broker for distribution to the querying components. The step related to data cleansing takes place before the RDFization and right after receiving sensors input. The module that addresses this part is optional and relies on statistical models such as min-max value based outlier analysis. After that, the Smart'Ops read compressed RDF events from Kafka topics, store them to create data sets (windows) and, once an execution point (step) has been reached, evaluate a configured SPARQL query against the dataset. Query results produce new (compressed RDF) events that are outputted to Kafka to be processed by another Smart'Op, archived or displayed to the end-user by a visualization module in which the goal is to ease the interpretation of analyzed streams through different forms of graphics. WAVES comes with a JAVA API[6] to help developers make desired extensions. Finally, the end-user could easily parametrize the system based on an RDF configuration file which includes parameters such as the window size, step, continuous query, etc.

### 3.1 Event modeling and Time reference

WAVES, as an RDF streaming processing (RSP) engine, models data streams as an unlimited flow of events where each event is a set of RDF triples associated to a timestamp. The overall structure of an event in WAVES represents the observation recorded by the sensor and associated with a specific timestamp. Opposite to many RSP engines, WAVES does not consider events as a set of independent timestamped RDF triples but as an atom (timestamp graph) that can not be split. Hence, WAVES evaluates the continuous query against whole events in the window. This guarantees output consistency as no event fails to match the SELECT clause because some of its triples are missing from the current evaluation window, as this may occur in other RSP engines due to throughput issues [7].

While WAVES was designed as a real-time stream processing engine consuming sensor measures as they are being produced, many use cases mandate processing past data recorded on files. For such cases, WAVES does not use the current system clock but its own time reference. This time reference is a distributed, synchronized, shifted and accelerated clock; shifted to align WAVES clock on the timestamp of the events read from files and accelerated so that WAVES can process months of legacy data within minutes. Using this time reference, WAVES can synchronize the reading of timestamped data from many input files distributed on a cluster so that events that occurred at the same moment in the past are processed at the same time.

### 3.2 Kafka Message Broker and Storm-based Smart'OP

The Kafka component is becoming a de-facto standard in stream processing engines and can be connected to most open-source streaming engines such as

[6] WAVES Java API available at `http://waves-rsp.org/api/index.html`

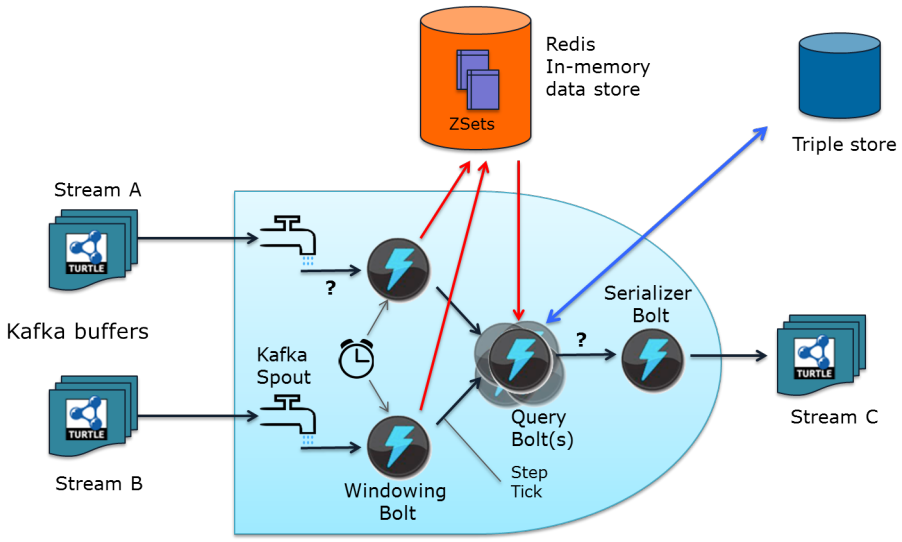



**Fig. 2:** Waves Smart'OP

Apache Spark Streaming and Apache Storm. It acts as a distributed message broker, based on a publish-subscribe approach.

Events are fetched from Kafka by a Storm-based Smart'Op component, which consists of a set of distributed nodes implemented by a Storm topology, i.e. a network of the so-called spouts and bolts. A spout is the source of stream and can read data from an external source such as a Kafka topic. A bolt is a processing logic unit which can perform any kind of processing such as filtering, joining, aggregation or interacting with external data stores. Each spout or bolt executes as tasks distributed across a Storm cluster, each task corresponds to one thread of execution. Topologies execute across one or more worker, each worker being a physical process running the Java Virtual Machine (JVM) and executing a subset of all the tasks for the topology.

WAVES was designed to be a fault tolerant platform that enables the system to continue operating properly when some of its components fail. Storm as a stream processing engine guarantees that the processing runs infinitely, automatically restarting any workers if there are faults. The core of Storm is an acking mechanism that can efficiently determine when a source tuple has been fully processed. Storm will replay tuples that fail to be fully processed within a topology-configurable timeout. So the core mechanisms in Storm provide an at-least-once processing guarantee for data.

In a WAVES topology, each spout subscribes to a single events stream represented by a Kafka topic. As many spouts as streams taking part in query evaluation are needed. For each stream, a windowing bolt is in charge of storing received events until a execution point (processing step) is reached. Due to the distributed nature of WAVES topologies, a need for shared storage arose.

WAVES relies on Redis in-memory store that natively supports rich data structures (such as sorted sets) and range queries as well as replication and periodic on-disk persistence. Once a processing step has been reached, the query bolt gathers the events in the current window for each input stream from the store, uncompresses them to populate an in-memory RDF store and triggers the execution of the SPARQL query. The query results, if any, form the resulting event for this step. After RDF encoding and compression, the Serializer bolt writes this event into the Kafka topic associated to the output stream of the topology.

## 4 Stream Compression in Distributed Context (D-RDSZ)

A distributed architecture for semantic reasoning requires high scalability and low latency to cope with massive real-time streams. However, frequent data transfers between several components (e.g., messaging middleware, data stores, etc.) produce significant overhead on the network and could affect the bandwidth. There are various methods to deal with the complex issue of network overhead. The one we will focus on is data compression, since RDF has a verbose data structure. Being able to reduce the size of each RDF transfer is of critical importance within a distributed architecture. As a solution, Garcia et al. [6] proposed a new approach called RDSZ which takes the stance that events in a given stream share structural similarities, i.e., the shape of the RDF graphs are more or less the same. A new graph (e.g., set of RDF triplets, RDF file, RDF event) can be represented on the basis of the previous graph. The main principle is to break up each graph into two parts, namely the Pattern and the Bindings. For each incoming graph, the subjects and the objects are replaced by variables x0, x1,...xn. The pattern represents the relationships (i.e. RDF properties) between these variables, and the bindings represent the correspondences between each variable and its value. Once the pattern extracted, the system checks if it exists in a cache or not. If the pattern exists, its identifier is recovered and associated with the bindings of the current graph. Otherwise, the pattern is attributed a new identifier and stored in the cache. This ensures that each graph pattern is stored only once in the cache and attributed a unique identifier. Then, the bindings of the current graph $N$ are compressed using a differential approach based on the bindings of the previous graph $N-1$. If the graph $N$ shares bindings with the graph $N-1$, then they are replaced by blanks. Finally, only the bindings and the pattern identifier are sent in the stream compressed using zlib[7] which exploits additional redundancies to compress even more. Using the RDSZ approach enables us to reduce the data streams up to 60% on average in our experiments, which is already a significant rate.

In this work, we propose an extension of RDSZ to improve compression and to adapt it for a distributed architecture. Our approach called D-RDSZ (Distributed RDSZ), uses gzip[8] instead of zlib. Introducing gzip scales down the initial data amount more than zlib in our experiments. Then, we propose to reduce the

[7] `http://www.zlib.net/`
[8] `http://www.gzip.org/`

URLs, which are long chains by introducing the namespaces. With each pattern, we associated the list of prefixes and related namespaces. As a result, the new URLs in the bindings appear much shorter than the original URLs, leading to higher compression rate than the original RDSZ. Listing 1 represents an example of bindings in our dataset after the D-RDSZ compression.

```
0- waves:event_1j_sh
1- waves:obs_1j_sh
2- waves:Q_DT01
3- "2015-01-01T01:15:00"^^xsd:dateTime
4- ssn:SensorOutput
5- "1.3E-1"^^xsd:double
6- ssn:ObservationValue
```

**Listing 1:** Example of Bindings after D-RDSZ compression

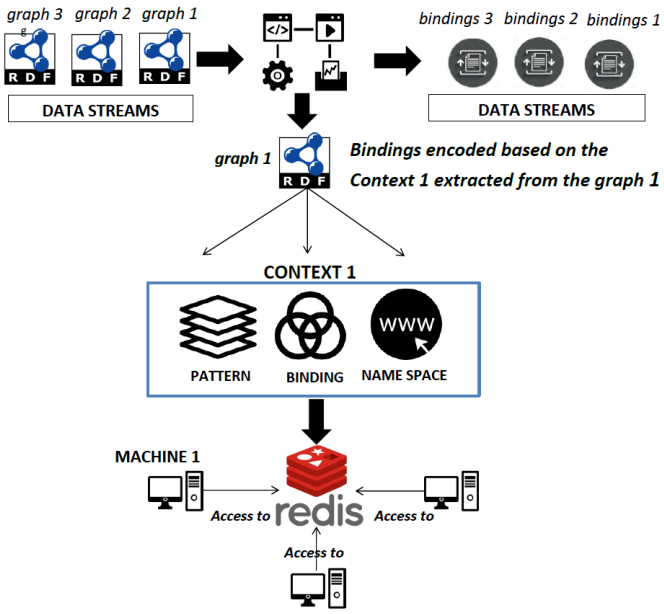



**Fig. 3:** D-RDSZ compression in distributed environment

In addition, as for RDSZ, the mechanism is still not adapted for distributed architecture since encoding the current graph $N$ is based on the bindings of the previously processed graph $N-1$. This requires data exchange between distributed machines if the graphs $N$ and $N-1$ are processed in different nodes,

thus leading to a network overhead. To solve this problem, we propose to encode the current graph $N$ based on the bindings of the initially processed graph from which the pattern has been extracted and stored. Hence, we create the so-called D-RDSZ context with which we associate the pattern, the bindings of the graph from which the pattern has been extracted and the prefixes. All the incoming graphs are encoded based on the context with which they share the same pattern. To guarantee the access to contexts in a distributed environment, we need to store them in a centralized system. Redis has been chosen for storage due to its convenient features (e.g key-value in-memory store, fast read/write, etc.). Each context created is automatically stored in Redis as shown in Figure 3. Since the contexts are stored in a centralized system, all the machines should have access to compress and decompress RDF graphs. In addition, each machine benefits from its local cache LRU (Least Recently Used). That is each machine contains a cache with latest recently used patterns to speed up the read access.

## 5 Experiments

### 5.1 Evaluation Scenario

For evaluation, we use a real world dataset describing different water measurements captured by sensors spread throughout the underground water pipeline system. Values of flow, pressure and chlorine are examples of these measurements. Basically, the set-up is built upon two queries with ascending complexity to evaluate the behavior of each engine under varying environments. More precisely, we use the following two queries:

- *Query 1:* It returns the list of sensors that have measures between 5 and 12, along with the observation timestamp.
- *Query 2:* It corresponds to the overall consumption represented by the sum of input flow grouped by the observation timestamp.

For each of the above queries, we parametrized the window size (2 seconds and 4 seconds) and the step (2 seconds and 1 second resp.) thus evaluating respectively tumbling windows and sliding windows. Concerning the amount of input streams, it is controlled by the number of sensors configured in the stream generator, in which the interval between each sensor measurements is set to 1 second. We replicated each experiment 5 times to gain consistent results and illustrate the distribution of result metrics obtained for (i) precision, recall to evaluate output correctness; and (ii) execution time to evaluate performance.

### 5.2 Evaluation Metrics

- *Stream Compression:* The stream compression has been evaluated by measuring the ratio between the uncompressed size and compressed size.
- *Oracle Metrics:* To check the correctness of query results, we used an oracle that determines the validity (i.e., correct or incorrect) of the query results by measuring the precision and recall. To achive this, we have used the oracle proposed by the YABench RSP benchmark [7].

– *System Performance:* To check the effectiveness of our distribution model based on Apache storm, we have implemented a measurement toolkit which analyzes thoroughly the behavior of the WAVES platform. To evaluate the performance, we have measured the average execution time to run queries.

### 5.3 Stream Compression

Evaluating the D-RDSZ approach on our dataset produces the results shown in Table 1. We can observe that applying gzip on streaming graphs increases the compression rate by 15% compared with zlib. The addition of namespaces improved this rate by 22% on average. We also observe that data compression is very fast where the time to compress a single event is about 0.006 milliseconds. These results attest the effectiveness of our D-RDSZ approach to improve compression and makes it feasible in distributed environment.

| File number | Size | RDSZ | RDSZ+gzip | RDSZ+Ns | D-RDSZ (RDSZ+gzip+Ns) |
|---|---|---|---|---|---|
| 1 | 1.3 Kb | 84.3% | 89.2% | 92.6% | 93.5% |
| 1000 | 1.4 Gb | 62.4% | 78.3% | 72.4% | 84.6% |

**Table 1:** Compression results for one file and for 1000 files –Ns stands for Namespaces-

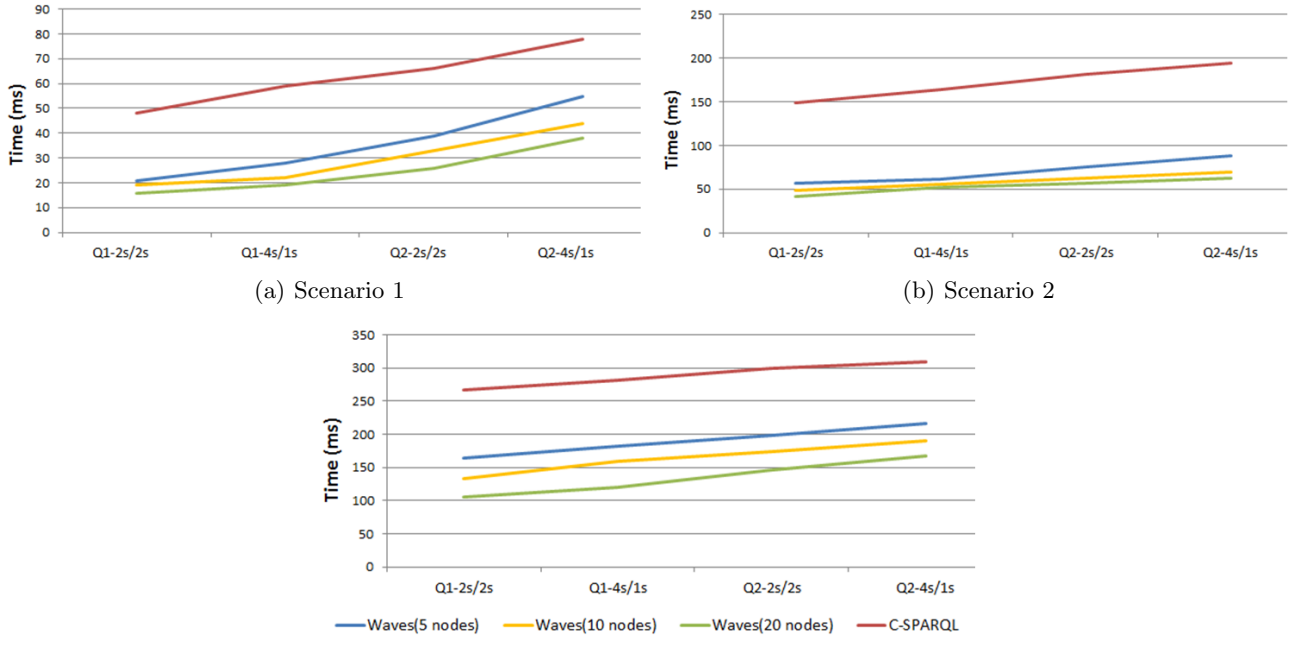
### 5.4 Precision and Recall

In this experiment, we compare our system with the conventional centralized version of C-SPARQL engine. Even though C-SPARQL is not tailored towards a distributed environment, we chose it to make the comparison due to the lack of available and ready-to-use distributed RSP engines. According to certain benchmarks [7, 4], C-SPARQL is known to be a popular RSP engine that delivers correct results even under high input load compared with existing engines. Table 2 shows the results of precision and recall for each of the three load scenarios (small: s= 10 sensors, medium: s= 50 sensors, and high: s= 100 sensors). The complexity of the scenarios is in the ascending order, from the least complex configuration (i.e. scenario 1) that loads roughly 1.500 triples, to the most complex configuration (i.e. scenario 3) that injects more than 20.000 triples. We can observe that WAVES succeeds to maintain 100% precision and recall under small and medium load (i.e. s = 10 sensors and s = 50 sensors) whereas C-SPARQL achieves slightly lower values (precision is at 100% only under small load). Generally, we observe that recall is lower than precision for C-SPARQL, where the values drop down dramatically when the engine is put under big load (e.g. s = 100 sensors). WAVES remains robust through all the scenarios with values above 75% for precision and recall.

| | | (a) Scenario1 | | (b) Scenario 2 | | (c) Scenario 3 | |
|---|---|---|---|---|---|---|---|
| | | WAVES | C-SPARQL | WAVES | C-SPARQL | WAVES | C-SPARQL |
| Precision | Q1-2s/2s | **100%** | **100%** | **100%** | 94% | **98%** | 80% |
| | Q1-4s/1s | **100%** | **100%** | **100%** | 88% | **84%** | 78% |
| Recall | Q2-2s/2s | **100%** | 93% | **97%** | 95% | **79%** | 56% |
| | Q2-4s/1s | **100%** | 91% | **94%** | 84% | **72%** | 43% |

**Table 2:** Precision/recall results for WAVES and C-SPARQL for (a) s= 10 sensors, (b) s= 50 sensors and (c) s= 100 sensors.

### 5.5 System Performance




(a) Scenario 1

(b) Scenario 2

(c) Scenario 3

**Fig. 4:** Q1 and Q2 execution time for the 3 scenarios: (a) s= 10 sensors, (b) s= 50 sensors and (c) s= 100 sensors.

To assess the scalability performance of WAVES, we use a metrics toolkit based on the Dropwizard library [1]. By varying the number of EC2 instances that run the apache Storm engine, we can confirm our distribution model effectiveness.

The experimentation is based on three different set-ups by increasing the number of nodes: 5 nodes, 10 nodes and 20 nodes. The graphs on Figure 4 indicate the execution time with indicators for the windows for different input load (small: s= 10 sensors, medium: s= 50 sensors, and high: s= 100 sensors). We found that WAVES is generally two to three time faster than C-SPARQL with a 5 nodes distribution model. However, the gap diminishes slightly under an important load (i.e. s= 100 sensors) due to the notable set-up time the nodes take to process, gather and send the results. It is also noticeable that WAVES system shows the best overall results under medium distribution (i.e. 10 nodes), therefore the overall performance gain for the 20 nodes set-up is not that significant.

## 6 Conclusion

In this paper, we have presented the design of a distributed and production-ready RDF stream processing engine. Our generic architecture is based on the use of popular big data frameworks which have proven to be useful in countless scientific and industrial applications. The WAVES system is able to run continuous queries on event-based streaming data in distributed environment. Moreover, it is practical for various applications as the end-user needs just to manipulate few parameters in the configuration file such as window size, step, query and parallelism degree. Finally, the evaluation results attest the performance and the robustness of our system. In the future, we plan to enable querying directly over compressed streams thus we do not need to decompress data which may improve the performance. We also aim to extend the architecture by introducing novel successful distributed frameworks such as Spark Streaming and Flink which have recently proven to be effective for stream processing.

## Acknowledgments

This work has been supported by the WAVES project which is partially supported by the French FUI (Fonds Unique Interministriel) call #17. The WAVES consortium is composed of industrial partners Atos, Ondeo Systems and Data publica, and academic partners ISEP and UPEM.

## References

1. Metrics Dropwizard, 2010-2014. `http://metrics.dropwizard.io/3.1.0/`.
2. Redis, Dec 2015. `http://http://redis.io/`.
3. D. F. Barbieri, D. Braga, S. Ceri, E. Della Valle, and M. Grossniklaus. C-SPARQL: SPARQL for continuous querying. In *WWW '09: Proceedings of the 18th international conference on World wide web*, pages 1061–1062, 2009.
4. D. Dell'Aglio, J. P. Calbimonte, M. Balduini, O. Corcho, and E. D. Valle. On correctness in rdf stream processor benchmarking. In *The 12th International Semantic Web Conference (ISWC2013)*, pages 321–336, 2013.

5. L. Fischer, T. Scharrenbach, and A. Bernstein. Scalable linked data stream processing via network-aware workload scheduling. In *Proceedings of the 9th International Workshop on Scalable Semantic Web Knowledge Base Systems*, pages 81–96, 2013.
6. N. F. García, J. Arias-Fisteus, L. Sánchez, D. Fuentes-Lorenzo, and Ó. Corcho. RDSZ: an approach for lossless RDF stream compression. In *The Semantic Web: Trends and Challenges - 11th International Conference, ESWC 2014, Anissaras, Crete, Greece, May 25-29, 2014*, pages 52–67.
7. M. Kolchin, P. Wetz, E. Kiesling, and A. M. Tjoa. Yabench: A comprehensive framework for RDF stream processor correctness and performance assessment. In *Web Engineering - 16th International Conference, ICWE, Lugano, Switzerland, June 6-9, 2016.*, pages 280–298, 2016.
8. D. Le-Phuoc, M. Dao-Tran, J. X. Parreira, and M. Hauswirth. A Native and Adaptive Approach for Unified Processing of Linked Streams and Linked Data. In *Proceedings of the 10th international conference on The Semantic Web - Volume Part I*, ISWC'11, pages 370–388. Springer-Verlag, 2011.
9. L. Neumeyer, B. Robbins, A. Nair, and A. Kesari. S4: Distributed stream computing platform. In *Proceedings of the 2010 IEEE International Conference on Data Mining Workshops*, pages 170–177, Washington, DC, USA, 2010. IEEE Computer Society.
10. D. L. Phuoc, H. N. M. Quoc, C. L. Van, and M. Hauswirth. Elastic and scalable processing of linked stream data in the cloud. In *The Semantic Web - ISWC 2013*, pages 280–297, 2013.
11. A. Sheth, C. Henson, and S. S. Sahoo. Semantic sensor web. *IEEE Internet Computing*, 12(4):78–83, July 2008.
12. A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, N. Bhagat, S. Mittal, and D. Ryaboy. Storm@Twitter. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 147–156, New York, NY, USA, 2014. ACM.
13. G. Wang, J. Koshy, S. Subramanian, K. Paramasivam, M. Zadeh, N. Narkhede, J. Rao, J. Kreps, and J. Stein. Building a replicated logging system with apache kafka. *Proc. VLDB Endow.*, 8:1654–1665, 2015.