# CityBench: A Configurable Benchmark to Evaluate RSP Engines using Smart City Datasets *

Muhammad Intizar Ali, Feng Gao and Alessandra Mileo

Insight Centre for Data Analytics,
National University of Ireland, Galway, Ireland
{ali.intizar,feng.gao,alessandra.mileo}@insight-centre.org

**Abstract.** With the growing popularity of Internet of Things (IoT) and IoT-enabled smart city applications, RDF stream processing (RSP) is gaining increasing attention in the Semantic Web community. As a result, several RSP engines have emerged, which are capable of processing semantically annotated data streams on the fly. Performance, correctness and technical soundness of few existing RSP engines have been evaluated in controlled settings using existing benchmarks like LSBench and SR-Bench. However, these benchmarks focus merely on features of the RSP query languages and engines, and do not consider dynamic application requirements and data-dependent properties such as changes in streaming rate during query execution or changes in application requirements over a period of time. This hinders wide adoption of RSP engines for real-time applications where data properties and application requirements play a key role and need to be characterised in their dynamic setting, such as in the smart city domain.
In this paper, we present CityBench, a comprehensive benchmarking suite to evaluate RSP engines within smart city applications and with smart city data. CityBench includes real-time IoT data streams generated from various sensors deployed within the city of Aarhus, Denmark. We provide a configurable testing infrastructure and a set of continuous queries covering a variety of data- and application- dependent characteristics and performance metrics, to be executed over RSP engines using CityBench datasets. We evaluate two state of the art RSP engines using our testbed and discuss our experimental results. This work can be used as a baseline to identify capabilities and limitations of existing RSP engines for smart city applications.

## 1 Introduction

Recent advances in Semantic Technologies for IoT have created great opportunities for rendering IoT-enabled services in smart cities. As a result, an increasing

* This research has been partially supported by Science Foundation Ireland (SFI) under grant No. SFI/12/RC/2289 and EU FP7 CityPulse Project under grant No.603095. http://www.ict-citypulse.eu

number of cities have started to invest in data-driven infrastructures and services for citizens, mostly focusing on creating and publishing a rich pool of dynamic datasets that can be used to create new services [8, 17]. Leveraging this data and semantic technologies, tools and solutions have been developed to abstract, integrate and process this distributed and heterogenous data sources.

One of the major aspects that captured the attention of the scientific community and standardisation bodies is the design of query languages, processes and tools to process RDF streams dynamically and in a scalable way[1]. Despite the success of RDF Stream Processing (RSP) solutions in this direction[14, 2, 1, 12, 4], available benchmarks for their evaluation are either synthetic or mostly based on static data dumps of considerable size that cannot be characterised and broken down[13, 18]. Few of the existing RSP engines have been evaluated using offline benchmarks such as SRBench and LSBench [13, 18, 6], but none of them has been tested based on features that are significant in real-time scenarios. There is a need for a systematic evaluation in a dynamic setting, where the environment in which data is being produced and the requirements of applications using it are dynamically changing, thus affecting key evaluation metrics.

In this paper, we distinguish different characteristics of benchmarking for RSP engines with a closer look to real-time requirements of smart city applications. We use real-time datasets from the city of Aarhus and present their schema, time-dependent features and interdependencies. We provide a testing environment together with a set of queries classified into different categories for evaluation of selected application scenarios. CityBench will prove as a tool for evaluating RSP engines within smart city applications based on their dynamic features (including performance), and comparing RSP engines in terms of their ability to fulfil application-specific requirements.

Our main contributions in this paper can be summarised as follows:

- we identify a set of dynamic requirements of smart applications which must be met by RSP engines;
- we design a benchmark based on such requirements, using realtime datasets gathered from sensors deployed within the City;
- we provide a configurable benchmarking infrastructure, which allows to set up evaluation tests enabling fine tuning of various configuration parameters;
- we provide a set of queries covering broader features of the RSP Query Languages in selected scenarios;
- finally, we evaluate state of the art RSP engines on our benchmark with different configurations, and we perform an empirical analysis of the experimental results.

**Structure of the paper:** Section 2 defines challenges and dynamic requirements for benchmarking of RSP engines in Smart City applications. We present the CityBench Benchmarking Suite in Section 3 and its evaluation in Section 4. Section 5 discusses state of the art, we conclude with final remarks in Section 6.

[1] `https://www.w3.org/community/rsp/` (l.a. Apr. 2015)

## 2 Smart City Applications: Challenges and Requirements for RSP

Challenges and requirements of smart city applications are inherently related to their dynamic nature and changing environment [9]. In this section, we identify various challenges (**Cn**) and respective requirements (**Rn**) which can potentially effect performance, scalability and correctness of smart city applications designed to query and integrate dynamic smart city datasets via RSP.

**C1: Data Distribution.** City data streams are instinctively distributed. Increase in the number of streams to be processed within a single query can have adverse effect over the performance of the engine. **R1:** RSP engines should be capable of addressing the challenge of high distribution and their performance should not be effected with higher degree of distribution.

**C2: Unpredictable Data Arrival Rate.** Data streams originated from sensor observation are mostly generated at a fixed rate. For example, a temperature sensor can be easily configured to produce each observation after a certain time period. Event data streams instead produce data at a variable rate and the observation rate for events is dependent upon the detection of a query pattern representing the event. **R2:** Applications consuming aggregated data streams at variable rates (e.g. events) should be able to cope with sudden increases in the streaming rate. Such increase or *stream burst* can potentially compromise the performance of RSP engines.

**C3: Number of Concurrent Queries.** Similar to the stream bursts, the number of users of an IoT-enabled smart city applications can suddenly increase. For example, a sudden increase of concurrent users of an application designed to monitor traffic conditions can be observed during traffic jams or accidents. **R3:** RSP engines should be stress tested by increasing the number of concurrent queries in their performance evaluation.

**C4: Integration with Background Data.** Some of the existing RSP engines have already demonstrated their capability to integrate background data. Executing queries over a larger size of such static background data may strongly affect the performance of RSP engine, and this aspect has not been thoroughly considered in current benchmarks. **R4:** RSP engines should be capable to deal with large amount of background data by applying proper data management techniques.

**C5: Handling Quasi-static Background Data.** Current RSP implementations load background data before query execution over static and dynamic data and cache it for longer periods. However, some of the background data can be quasi-static (e.g. changing with irregular periodicity) such as the number of follower of a twitter user, or the price of utilities. Materialising this data at query time is not efficient, but caching might result in out-of-date results. **R5:** RSP engines should be able to efficiently update the quasi-static background data during query execution using effective strategies to determine what data is more likely to be out-of-date.

**C6: On-demand Discovery of Data Streams.** In smart city environments, many applications do not have prior knowledge of the available streaming data sources that can potentially be relevant for a specific task. Therefore, discovering relevant streaming sources on the fly is a challenge. **R6:** RSP query languages should provide support for stream discovery and federation, possibly taking into account quality constraints so that the best available source is considered.

**C7: Adaptation in Stream Processing.** Smart city applications are operated over dynamic and distributed infrastructure, without any central control. This makes it difficult to provide efficient strategies for adapting to changing environments that are typical of smart city applications. For example availability of sensors, communication issues, changes in the environment or user needs can demand for the use of alternative data streams. **R7:** RSP solutions should be able to switch between multiple semantically equivalent data streams during query execution.

## 3 CityBench Benchmarking Suite

In this section, we present CityBench Benchmarking Suite consisting of, (i) *Benchmark Datasets,* designed over realtime smart city datasets, (ii) *Configurable Testbed Infrastructure,* containing a set of tools for dataset preparation and testbed set-up to benchmark RSP engines, and (iii) *Queries*, a set of continuous queries covering the query features and challenges discussed in Section 2. In what follows, we discuss each of these three components.

### 3.1 Benchmark Datasets

Leveraging the outcomes of the CityPulse project[2], we use the dataset collected from the city of Aarhus, Denmark[3,4]. In this section, we briefly describe each of the dataset in the benchmark and elaborate on the semantic representation of the datasets.

**Vehicle Traffic Dataset.** This dataset contains traffic data. The City administration has deployed 449 pairs of sensors over the major roads in the city. Traffic data is collected by observing the vehicle count between two points over a duration of time. Observations are currently generated every five minutes. A meta-data dataset is also provided which contains information about location of each traffic sensor, distance between one pair of sensors and type of road where the sensors have been deployed. Each pair of traffic sensors reports the average

[2] http://www.ict-citypulse.eu/

[3] We acknowledge the CityPulse consortium team for the provision of Datasets `http://iot.ee.surrey.ac.uk:8080/datasets.html` (l.a. Apr. 2015)

[4] CityBench datasets are made publicly available by EU Project CityPulse, for use of any part of these datasets, the source must be properly acknowledged. The accuracy or reliability of the data is not guaranteed or warranted in any way.

vehicle speed, vehicle count, estimated travel time and congestion level between the two points set over a segment of road.

**Parking Dataset.** Parking lots in Aarhus are equipped with sensors and capable of producing live data streams indicating number of vacant places. The Parking Dataset consists of observations generated by 8 public parking lots around the city.

**Weather Dataset.** Currently, there is only a single weather sensor available in the city to collect live sensor observations about the weather condition. Weather sensor data provides observations related to dew point (°C), humidity (%), air pressure (mBar), temperature (°C), wind direction (°), and wind speed (kph).

**Pollution Dataset.** Pollution is directly related to the traffic level, however due to unavailability of the pollution sensors, a synthesised pollution data for the city of Aarhus is made available to complement the traffic dataset. Observation points for traffic sensors (446) are replicated to create mock-up sensors for pollution at the exact same location as traffic sensors. An observation for air quality index is generated every 5 minutes using a pre-selected pattern. Details regarding the procedure followed to synthesised pollution data are accessible at: `http://iot.ee.surrey.ac.uk:8080/datasets/pollution/readme.txt`.

**Cultural Event Dataset.** This dataset is quasi-static and contains cultural events provided by the municipality of Aarhus. The dataset is periodically updated to reflect the latest information related to planned cultural events,. Updates are available as data stream (a notification service notify of any updates in the dataset). However, due to the low frequency of updates, we consider this dataset as background knowledge and use it to demonstrate integration of the static data with the data streams.

**Library Events Data.** This dataset contains a collection of past and future library events hosted by libraries in the city. A total collection of 1548 events is described in this dataset. Similarly to the Cultural Events Dataset, updates in the Library Events Dataset are also not frequent, therefore the dataset is considered quasi-static.

**User Location Stream.** Most of the IoT-enabled smart city application are designed to be location-aware, therefore they strongly rely over updates on the location of mobile users. We synthesised a User Location Stream to mock-up a real usecase scenario of users roaming around. This data stream contains periodic observations with geo-location coordinates of a fictional mobile user.

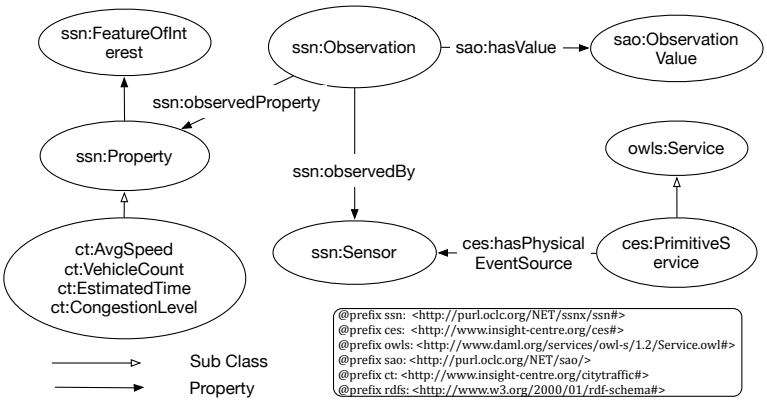


**Fig. 1.** Ontological Representation of Traffic Sensor Observation

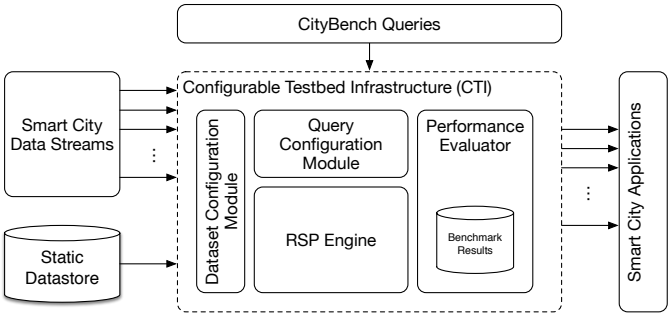


**Fig. 2.** An Overview of the Configurable Testbed Infrastructure

Users of CityBench can download the existing as well as any future datasets from the CityBench website[5]. All of the above mentioned datasets are semantically annotated and interlinked using the CityPulse information model[6]. The SSN Ontology [5] is a de-facto standard for sensor observation representation, and it is also a central part of the CityPulse Information Model. Figure 1 shows a sample representation of traffic sensor observations semantically annotated using the SSN ontology.

### 3.2 Configurable Testbed Infrastructure

As discussed in Section 2, performance of RSP engines does not depend only on language features but also on dynamic metrics related to the data and to the ap-

[5] `https://github.com/CityBench/Benchmark`
[6] `http://iot.ee.surrey.ac.uk:8080/info.html`

plication. To evaluate the performance of RSP engines according to the dynamic requirements of smart city applications, we provide a *Configurable Testbed Infrastructure* (CTI) containing a set of Application Programming Interface (API's) to set up the testbed environment [7]. CTI allows its users to configure a variety of metrics for the evaluation of RSP engine. Figure 2 provides an overview of the CTI, there are three main modules, (i) *Dataset Configuration Module:* allows configuration of stream related metrics, (ii) *Query Configuration Module:* allows configuration of query related metrics, and (iii) *Performance Evaluator:* is responsible for recording and storing the measurements of the performance metrics. We discuss configuration metrics in what follows.

**Changes in Input Streaming Rate:** The throughput for data stream generation can be configured in CityBench. For example, a rate $r \in [1, inf]$ can be configured to set up the streaming rate to the real interval between observations ($r = 1$ means replay at original rate), or a frequency $f$ can be used to set a different streaming rate.

**PlayBack Time:** CityBench also allows to playback data from any given time period to replay and mock-up the exact situation during that period.

**Variable Background Data Size:** CityBench allows to specify which dataset to use as background knowledge, in order to test the performance of RSP engines with different static datasets. We also provide duplicated versions (with varying size) of two static datasets, *Cultural Event Dataset* and *Library Event Dataset*. Any version of the given background datasets can be loaded to test RSP engines with different size of background data.

**Number of Concurrent Queries:** CityBench provides the ability to specify any number of queries to be deployed for testing purposes. For example, any number of queries can be selected to be executed concurrently any number of times. Such situation will simulate a situation where a number of simultaneous users are executing the same query using any application.

**Increase in the Number of Sensor Streams within a Single Query:** In order to test the capability of the RSP engine to deal with data distribution, CityBench makes it possible to configure various size of streams involved within a single query. We achieved this by increasing the number of streams to be observed as relevant for the query. Query similar to traffic condition monitoring over a given path are best candidates for distribution test and number of streams involved within a query can be increased by simply increasing the length of the observed path.

[7] `https://github.com/CityBench/Benchmark/tree/master`

**Selection of RSP Query Engines:** CityBench allows to seamlessly use different query engines as part of the testing environment. Currently, we support CQELS and C-SPARQL. However, we encourage users to extend the list of RSP engines by embedding the engine within CTI.

### 3.3 Smart City Applications Queries over CityBench Datasets

In this section, we present a set of 13 queries covering most of the features and challenges discussed in Section 2. Our main goal while designing the queries is to highlight and evaluate the characteristics and features of the RSP engines which are most relevant to the smart city applications requirements. Benchmark queries designed to cover query specific features of the RSP engines can be found in the state of the art [18, 13, 6]. In what follows we identify three smart city applications from the CityPulse scenarios[8] and generate queries which are relevant for applications deployed around these scenarios.

**Multi-modal Context-aware Travel Planner:** This application relies on modules that can provide one or more alternative paths for users to reach a particular location. On top of these modules, the application aims at dynamically optimising users' path based on their preferences on route type, health and travel cost. In addition to that, the application continuously monitors factors and events that can impact this optimisation (including traffic, weather, parking availability and so on) to promptly adapt to provide the best up-to-date option. Relevant queries for this application scenario are listed below.

*Q1: What is the traffic congestion level on each road of my planned journey?*

This query monitors the traffic congestion from all traffic sensors located on the roads which are part of the planned journey.

*Q2: What is the traffic congestion level and weather conditions on each road of my planned journey?*

*Q2* is similar to *Q1* with an additional type of input streams containing weather observations for each road at the planned journey of the user.

*Q3: What is the average congestion level and estimated travel time to my destination?*

This query includes the use of aggregate functions and evaluates the average congestion level on all the roads of the planned journey to calculate the estimated travel time.

*Q4: Which cultural event happenig now is closest to my current location?*

*Q4* consumes user location data streams and integrates it with background knowledge on the list of cultural events to find out the closest cultural event happening near his current location.

[8] `http://www.ict-citypulse.eu/scenarios/`

*Q5: What is traffic congestion level on the road where a given cultural event X is happening? Notification for congestion level should be generated every minute starting from 10 minutes before the event X is planned to end, till 10 minutes after.*

*Q5* is a conditional query which should be deployed at the occurrence of an event and have predefined execution duration.

**Parking Space Finder Application:** This application is designed to facilitate car drivers in finding a parking spot combining parking data streams and predicted parking availability based on historical patterns. Additional sources such as timed no parking zones, congested hot spots and walking time from parking to a point of interest, the user can reduce circulation time and optimise parking management in the city. Queries related to this application are listed below.

*Q6: What are the current parking conditions within range of 1 km from my current location?*

This query represents a most common query issued by users of a parking application to easily find a nearby parking place.

*Q7: Notify me whenever a parking place near to my destination is full.*

*Q7* is a combination of travel planner and parking application, where a user wants to be notified about parking situation close to the destination.

*Q8: Which parking places are available nearby library event X?*

This query combines parking data streams with the static information about the library events to locate parking spaces nearby the library.

*Q9: What is the parking availability status nearby the city event with the cheapest tickets price?*

Similarly to *Q8*, this query monitors parking availability near a city event which has the cheapest ticket price.

**Smart City Administration Console:** This application facilitates city administrators by notifying them on the occurrence of specific events of interest. The dashboard relies on data analytics and visualisation to support early detection of any unexpected situation within the city and takes immediate actions, but it can also be used as a city observatory for analysing trends and behaviours as they happen. Queries related to this application are listed below.

*Q10: Notify me every 10 minutes, about the most polluted area in the city.*

*Q10* is an analytical query executed over the pollution data streams to find out which area in the city in most polluted and how this information evolves.

*Q11: Notify me whenever no observation from weather sensors have been generated in the last 10 minutes.*

This query helps to detect any faulty sensors which are not generating observations or networking issues.

*Q12: Notify me whenever the congestion level on a given road goes beyond a predefined threshold more than 3 times within the last 20 minutes.*

This query helps in early detection of areas where traffic conditions are becoming problematic.

*Q13: Increase the observation monitoring rate of traffic congestion if it surpasses a pre-specified threshold.*

This query provides a more frequent status update on congestion levels in critical conditions such as traffic jams or accidents.

CityBench provides all 13 queries ready to execute over CQELS and C-SPARQL, which can be downloaded from CityBench website[9].

## 4 Experimental Evaluation and Empirical Analysis

In order to showcase the feasibility of CityBench and highlight the importance of configuration parameters, we conducted our experimental evaluation over CQELS and C-SPARQL engines using CityBench Benchmarking Suite. We set up a testbed with multiple configuration of *CTI* performance metrics[10,11]. We evaluated the two RSP engines with respect to (i) Latency, (ii) Memory Consumption, and (iii) Completeness. The experiments in this paper covers requirements **R1** to **R4** (see Section 2). However there is no existing RSP engines which can meet **R5** to **R7**. It is worth mentioning that the overhead caused by the benchmark is insignificant and does not pose threats the validity of the results, i.e., for latency it costs several milliseconds to annotate a CSV row as a RDF graph, for memory consumption the benchmark uses up to 10 MB for tracking the results produced, for completeness the benchmark do not introduce any overhead.

### 4.1 Latency

Latency refers to the time consumed by the RSP engine between the input arrival and output generation. We evaluate the latency of RSP engines by increasing the number of input streams within a query and by increasing the number of concurrent queries executed.

[9] `https://github.com/CityBench/Benchmark/`

[10] Experiments are reproducible using *CTI* over CityBench Datasets, details are available at: `https://github.com/CityBench/Benchmark/`

[11] All experiments are carried out on a Macbook Pro with a 2.53 GHz duo core cpu and 4 GB 1067 MHz memory.

**Increasing the Number of Input Streams.** We designed three variations of query *Q10*[12] to generate an immediate notification about polluted areas in the city with three configurations for number of input data streams (2, 5 and 8). Results shown in Figure 3 depict that the overhead for C-SPARQL was minimal with increasing number of streams, however CQELS suffer from abnormal behaviour for query with 5 input streams (secondary y-axis in Figure 3) and was unable to process 8 input streams within a single query.

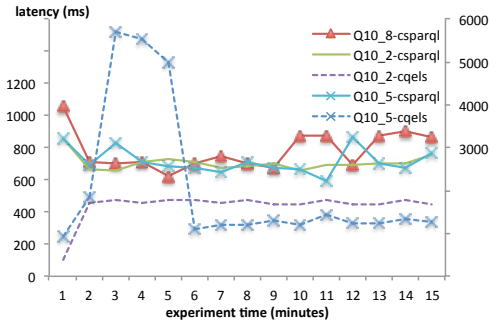


**Fig. 3.** Latency over Increasing Numer of Data Streams

**Increasing the Number of Concurrent Queries.** We performed our scalability test by executing *Q1*, *Q5* and *Q8* over both engines. Queries are executed with three different configuration (1, 10, and 20) for number of concurrent queries. Figure 4 and Figure 5 show the effect over latency with increasing number of concurrent queries for CQELS. A closer look at the results reveals that CQELS has a substantial delay, when the number of concurrent queries is increased from 0 to 10 for all three queries. However, CQELS performance is not much effected over subsequent increase from 10 to 20. As depicted in Figure 6 and Figure 7, C-SPARQL seems to have a constant size of overhead for delay with the increasing number of concurrent queries in contrast to CQELS.

### 4.2 Memory Consumption

We evaluated the two RSP engines by observing the usage of system memory during the concurrent execution of an increasing number of queries and increasing size of background data.

**Increasing the Number of Concurrent Queries.** We used query *Q1* and *Q5* and measure memory consumption during 15 minutes execution for each query. As shown in Figure 8, with an increasing number of concurrent queries, C-SPARQL has a minimal impact on memory consumption for both queries. However, with increasing duration for query execution, there is a constant increase in memory consumption for *Q5*, rate of increase in memory is similar for both single query execution and 20 concurrent queries execution. In contrast, CQELS seems to have increasing memory consumption issue for *Q1*, there is

[12] We selected different queries for each experiment based on their suitability for the corresponding configuration metric. A comprehensive report containing complete results for all queries is available at CityBench website: `https://github.com/CityBench/Benchmark/tree/master/result_log/samples`

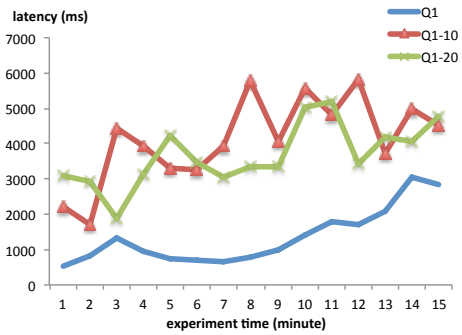


**Fig. 4.** Latency over Increasing Number of Concurrent Queries (*Q1* over CQELS)

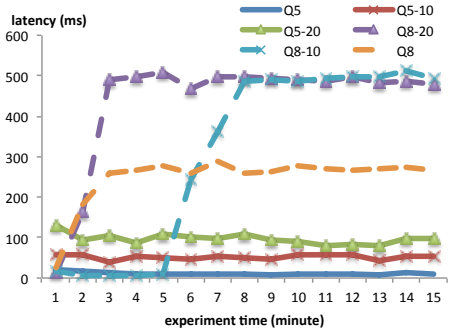


**Fig. 5.** Latency over Increasing Number of Concurrent Queries (*Q5* and *Q8* over CQELS)

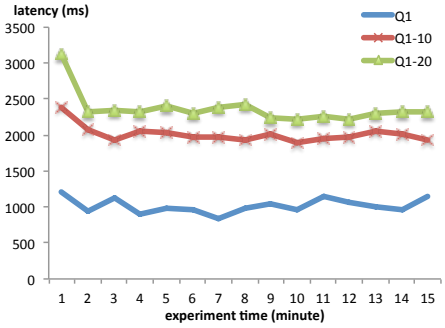


**Fig. 6.** Latency over Increasing Number of Concurrent Queries (*Q1* over C-SPARQL)

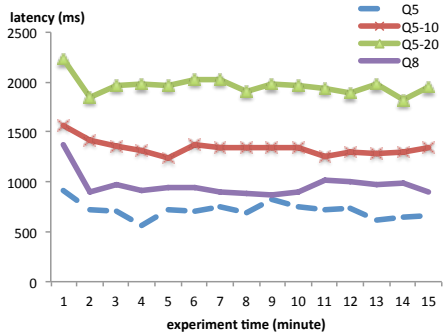


**Fig. 7.** Latency over Increasing Number of Concurrent Queries (*Q5* and *Q8* over C-SPARQL)

also a substantial increase in memory consumption for *Q1* after an increase in the number of concurrent queries from 1 to 20. As depicted in Figure 9, CQELS has better performance regarding the stability of the engine over the time period of 15 minutes execution of *Q5*. Also, it is noticeable that the memory consumption of *Q5* increases linearly and it would eventually reach the memory limit and crash the engine. The reason of the abnormal behaviour is perhaps the cross-product join on the static data in *Q5* creates a lot (millions) of intermediate results and are not cleared from the cache properly in both engines.

**Increasing the Size of Background Data.** We analysed memory consumption while increasing the size of background data. We generated three versions of background data required for the execution of query *Q5*, increasing the size from 3MB to 20MB and 30 MB. Figure 10 shows that CQELS seems to be better at memory management with background data of increasing size.

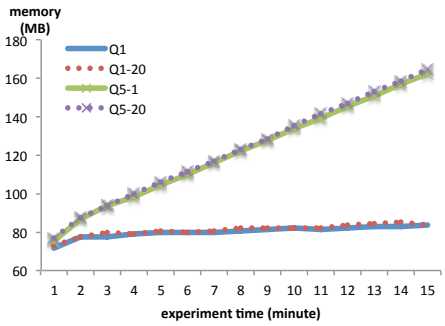


**Fig. 8.** Memory Consumption for Increasing Number of Concurrent Queries (C-SPARQL)

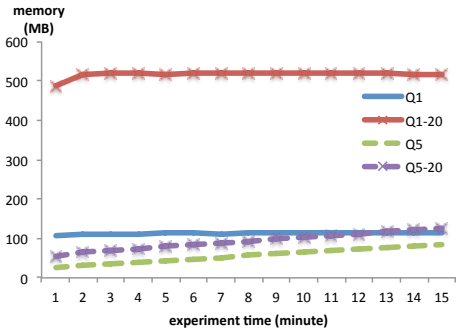


**Fig. 9.** Memory Consumption for Increasing Number of Concurrent Queries (CQELS)

### 4.3 Completeness

We evaluated the completeness of results generated by RSP engines by executing Query *Q1* with variable input rate of data streams. We allow each stream to produce $x$ observations and count $y$ unique observation IDs in the results, hence we have the completeness $c = y/x$. Note that we don't stop the streams immediately when they finished sending triples but waited for a period of time until no new results are generated, this ensured that the stream engines have enough time for query processing. Figure 11, shows that CQELS completeness level has been dropped up to 50%, while C-SPARQL continue to produce results with a completeness ratio of above 95%. The most probable cause of the completeness drop in CQELS is the complexity and concurrency of join over multiple streams.

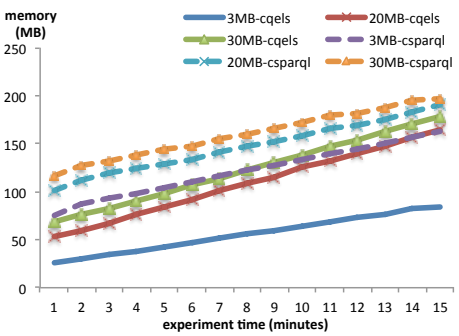


**Fig. 10.** Memory Consumption for Increasing Size of Background Data (*Q5*)

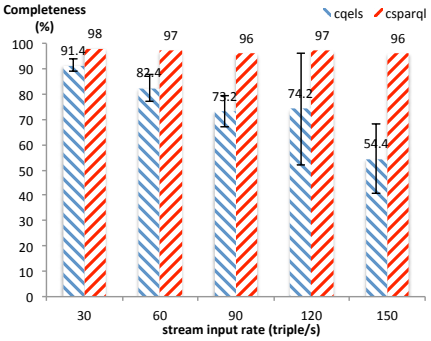


**Fig. 11.** Completeness of Results with Increasing Rate of Input Stream

## 5 Related Work

With advances in the use of semantic technologies to process linked stream data, tools and systems for RSP started to use SPARQL benchmarking systems to test their applications. There are 3 main prominent efforts in this area [3, 11, 16]. The Berlin SPARQL Benchmark is the most widely used RDF benchmarking system which portrays a usecase scenario of e-commerce. Lehigh university benchmark is designed to test OWL reasoning and inferencing capabilities over a university domain ontology. $SP^2$ benchmark uses DBLP data[13]. All of these SPARQL benchmarks are inspired by traditional approaches for benchmarking relational database management systems.

Understanding the different requirements and evaluation parameters needed for RDF data, new benchmarks specifically targeting the evaluation of RSP engines have been proposed. LS Bench and SR Bench are two well known efforts for benchmarking RSP engines. SR Benchmark is defined on weather sensors observations collected by Kno.e.sis[14]. The dataset is part of the Linked Open Data Cloud and contains weather data collected since 2002[15]. All sensor observations are semantically annotated using the SSN ontology. Beside weather streams, SR contains two static datasets (GeoNames[16] and DBPedia [17]) for integration of streaming data with background knowledge. The benchmark contains verbal description of 17 queries covering stream flow, background integration and reasoning features. However, due to the lack of a common RDF stream query language, some of the queries are not supported by the existing engines and therefore cannot be executed.

LS Benchmark is a synthetically generated dataset on linked social data streams. The dataset contains 3 social data streams, namely (i) Location (GPS coordinates) stream of a social media user, (ii) stream of micro posts generated or liked by the user, and (iii) a stream of notification whenever a user uploads an image. LS Bench also provides a data generator to synthesised datasets of varying size. LS Bench contains 12 queries, covering processing of streaming data as well as background data integration.

Both LS and SR benchmarks focus on evaluating RSP engines to demonstrate their query language support, process query operators and performance in a preconfigured static testbed. Best practices to design a benchmark are discussed in [10, 15]. Real-world environment for the applications using RSP is however dynamic. In [7], authors have demonstrated that synthesised benchmark datasets do not portray the actual dataset requirements and therefore might produce unreliable results. CityBench extends the existing benchmarks and takes a new perspective on the evaluation of RSP engine which relies on the applications requirements and dynamicity of the environment to draw a picture that is closer to reality.

[13] `http://dblp.uni-trier.de/`
[14] `http://knoesis.wright.edu`
[15] `http://wiki.knoesis.org/index.php/LinkedSensorData`
[16] `http://datahub.io/dataset/geonames`
[17] `http://wiki.dbpedia.org/`

## 6 Concluding Remarks and Future Directions

CityBench is a benchmark for the evaluation of RSP solutions in real dynamic settings, with real city data. This work has been motivated by the need to benchmark RSP systems moving away from pre-configured static testbed towards a dynamic and configurable infrastructure (CTI). This comprehensive benchmarking suite includes not only streaming and static datasets but also semantic annotation tools, stream simulation capabilities, and a set of parameters that best represent the set of data- and application- dependent characteristics and performance metrics that are typical of smart city applications.

This work will serve as a baseline for the evaluation of RSP engines in real application scenarios, and can be extended to accommodate additional features and datasets. Our initial evaluation of CityBench suggests the requirements identified to characterise smart city applications using streaming data provide a richer set of dimensions to evaluate RSP engines. The ability to tune these dimensions offers interesting insights on how application requirements play a key role in comparing and choosing one RSP solution over another. There are substantial differences not only in the language features but also in the windows operator and processing implemented within existing RSP engines, which is also reflected in how such engines perform on CityBench under different configurations.

Motivated by the need for a better approach to RSP, standardisation activities within the W3C RSP WG [18] have identified the need to converge towards a unified model for producing, transmitting and continuously querying RDF Streams. We believe that requirements and results presented in this paper can help guiding the roadmap towards better RSP solutions for smart cities and beyond. We are currently actively engaging with the RSP community towards this goal.

## References


1. D. Anicic, P. Fodor, S. Rudolph, and N. Stojanovic. EP-SPARQL: a unified language for event processing and stream reasoning. In *Proc. of the WWW 2011*. ACM, 2011.
2. D. F. Barbieri, D. Braga, S. Ceri, E. Della Valle, and M. Grossniklaus. C-sparql: Sparql for continuous querying. In *Proc. of WWW*, pages 1061–1062. ACM, 2009.
3. C. Bizer and A. Schultz. Benchmarking the performance of storage systems that expose sparql endpoints. *World Wide Web Internet And Web Information Systems*, 2008.
4. A. Bolles, M. Grawunder, and J. Jacobi. Streaming sparql extending sparql to process data streams. In *Proc. of ESWC 2008*, pages 448–462, Berlin, Heidelberg, 2008. Springer-Verlag.
5. M. Compton, P. Barnaghi, L. Bermudez, R. Garcia-Castro, O. Corcho, S. Cox, J. Graybeal, M. Hauswirth, C. Henson, A. Herzog, V. Huang, K. Janowicz, W. D. Kelsey, D. L. Phuoc, L. Lefort, M. Leggieri, H. Neuhaus, A. Nikolov, K. Page, A. Passant, A. Sheth, and K. Taylor. The ssn ontology of the w3c semantic sensor


[18] `https://www.w3.org/community/rsp/`

network incubator group. *Web Semantics: Science, Services and Agents on the World Wide Web*, 17:25 – 32, 2012.
6. D. Dell'Aglio, J.-P. Calbimonte, M. Balduini, O. Corcho, and E. D. Valle. On correctness in rdf stream processor benchmarking. In *Proc. of ISWC 2013*, volume 8219, pages 326–342. Springer, 2013.
7. S. Duan, A. Kementsietsidis, K. Srinivas, and O. Udrea. Apples and oranges: a comparison of rdf benchmarks and real rdf datasets. In *Proc. of ACM SIGMOD 2011*, pages 145–156. ACM, 2011.
8. T. et al. Real time iot stream processing and large-scale data analytics for smart city applications, 2014. Poster presented at European Conference on Networks and Communications.
9. F. Gao, M. I. Ali, and A. Mileo. Semantic discovery and integration of urban data streams. In *Proc. of S4SC @ ISWC 2014*, pages 15–30, 2014.
10. J. Gray. *Benchmark handbook: for database and transaction processing systems*. Morgan Kaufmann Publishers Inc., 1992.
11. Y. Guo, Z. Pan, and J. Heflin. Lubm: A benchmark for owl knowledge base systems. *Web Semantics: Science, Services and Agents on the World Wide Web*, 3(2):158–182, 2005.
12. S. Komazec, D. Cerri, and D. Fensel. Sparkwave: continuous schema-enhanced pattern matching over RDF data streams. In *Proc. of DEBS 2012*, pages 58–68, 2012.
13. D. Le-Phuoc, M. Dao-Tran, M.-D. Pham, P. Boncz, T. Eiter, and M. Fink. Linked stream data processing engines: Facts and figures. In *Proc. of ISWC 2012*, pages 300–312. Springer, 2012.
14. D. Le-Phuoc, M. Dao-Tran, J. Xavier Parreira, and M. Hauswirth. A native and adaptive approach for unified processing of linked streams and linked data. *Proc. of ISWC 2011*, pages 370–388, 2011.
15. T. Scharrenbach, J. Urbani, A. Margara, E. Della Valle, and A. Bernstein. Seven commandments for benchmarking semantic flow processing systems. In *The Semantic Web: Semantics and Big Data*, pages 305–319. Springer, 2013.
16. M. Schmidt, T. Hornung, G. Lausen, and C. Pinkel. Spˆ 2bench: a sparql performance benchmark. In *Data Engineering, 2009. ICDE'09. IEEE 25th International Conference on*, pages 222–233. IEEE, 2009.
17. C. S. N. A. S. A. M. Stefan Bischof, Athanasios Karapantelakis and P. Barnaghi. Semantic modeling of smart city data. In *Proc. of the W3C Workshop on the Web of Things: Enablers and services for an open Web of Devices*, Berlin, Germany, June 2014. W3C.
18. Y. Zhang, M.-D. Pham, Ó. Corcho, and J.-P. Calbimonte. Srbench: A streaming rdf/sparql benchmark. In *Proc. of ISWC 2012*, pages 641–657, 2012.