7

PlanetData

Network of Excellence

FP7 – 257641

D1.2 Benchmarking RDF Storage Engines

Coordinator: Ying Zhang (CWI)
With contributions from: Pham Minh Duc(CWI), Fabian Groffen(CWI), Erietta Liarou (CWI), Peter Boncz (CWI), Martin Kersten (CWI), Jean Paul Calbimonte (UPM), Oscar Corcho (UPM)
1st Quality reviewer: Irini Fundulaki (FORTH)
2nd Quality reviewer: Andreea Gagiu (STI)

Deliverable nature:	Report (R)
Dissemination level: (Confidentiality)	Public (PU)
Contractual delivery date:	M18 (March 31, 2012)
Actual delivery date:	March 30, 2012
Version:	1.0
Total number of pages:	58
Keywords:	SRBench, Streaming RDF, benchmark, Linked Stream Data, $SPARQL_{Stream}$

Abstract

In this deliverable, we present version V1.0 of ***SRBench***, the *first* benchmark for Streaming RDF engines, designed in the context of Task 1.4 of PlanetData, completely based on real-world datasets. With the increasing problem of too much streaming data but not enough knowledge, researchers have set out for solutions in which Semantic Web technologies are adapted and extended for the publishing, sharing, analysing and understanding of such data. Various approaches are emerging. To help researchers and users to compare streaming RDF engines in a standardised application scenario, we propose SRBench, with which one can assess the abilities of a streaming RDF engine to cope with a broad range of use cases typically encountered in *real-world* scenarios. We offer a set of queries that cover the major aspects of streaming RDF engines, ranging from simple pattern matching queries to queries with complex reasoning tasks. To give a first baseline and illustrate the state of the art, we show results obtained from implementing SRBench using the $\text{SPARQL}_{\text{Stream}}$ query-processing engine developed by UPM.

Executive summary

In this deliverable, we present version V1.0 of ***SRBench***, the *first general purpose* Streaming RDF Benchmark, which has been designed from scratch (in the context of Task 1.4 of PlanetData) to assess the streaming RDF engines.

With the increasing problem of *"too much (streaming) data but not enough knowledge"* (Sheth et al., 2008), i.e., existing techniques are not sufficient to gain and derive information from the large amount of available data (Balazinska et al., 2007; Della Valle et al., 2009), researchers have set out for solutions in which Semantic Web technologies are adapted and extended for the publishing, sharing, analysing and understanding of the streaming data. Various approaches are emerging, e.g., C-SPARQL, $\text{SPARQL}_{\text{Stream}}$, StreamSPARQL and CQUEL. To help researchers and users to compare streaming RDF engines in a standardised application scenario, we propose SRBench, with which one can assess the abilities of a streaming RDF engine to cope with a broad range of use cases typically encountered in *real-world* scenarios. The design of SRBench is based on an extensive study of the state-of-the-art techniques in both the data stream management systems and the streaming RDF processing engines, and the existing RDF/SPARQL benchmarks. This ensures that we capture all important aspects of streaming RDF processing in the benchmark.

Duan et al. (2011) present an extensive study of the characteristics of real-world RDF data and generated (i.e., synthetic) benchmark RDF data. The study shows that existing benchmarks do not accurately predict the behaviour of RDF stores in realistic scenarios. Motivated by this study, we have purposely chosen to not generate synthetic data for the benchmark, but use real-world datasets instead. Thus, SRBench uses three real-world datasets from the Linked Open Data cloud, i.e., the LinkedSensorData dataset, the GeoNames RDF dataset and the DBpedia dataset. The LinkedSensorData dataset contains streaming RDF data collected from US weather stations. It is the first and largest sensor dataset in the Linked Open Data cloud and the CKAN portal. The locations of the stations are linked to the locations described by the GeoNames dataset. The GeoNames locations are, in turn, linked to objects described by the DBpedia dataset (one of the largest and most popularly used dataset in the LOD cloud) through the `owl:sameAs` property.

The goal of SRBench V1.0 is to evaluate the functional completeness of a streaming RDF engine. The benchmark contains a concise, yet comprehensive set of queries which covers the major aspects of streaming SPARQL query processing, ranging from simple pattern matching queries to queries with complex reasoning tasks. The main advantages of applying Semantic Web technologies on streaming data include providing better search facilities by adding semantics to the data, reasoning through ontologies, and integration with other data sets. The ability of a streaming RDF engine to process these distinctive features is accessed by the benchmark with queries that apply reasoning not only over the streaming sensor data, but also over the metadata and even other data sets in the Linked Open Data (LOD) cloud.

Finally, we have complemented our work on SRBench with a functional evaluation of the benchmark on the $\text{SPARQL}_{\text{Stream}}$ query-processing engine developed by the PlanetData partner UPM. The engine supports the streaming RDF query language, also called $\text{SPARQL}_{\text{Stream}}$, proposed by UPM. The evaluation shows that the functionality supported by $\text{SPARQL}_{\text{Stream}}$ is fairly complete. At the language level, it is able to express all benchmark queries easily and concisely. At the query processing level, some missing features have been discovered, for all of which preliminary code has been added for further development.

This deliverable is a continuation of the previous deliverable D1.1, in the sense that D1.1 has studied, among others, the characterisation mechanisms of the Linked Stream Data. In this deliverable, we continue working on Linked Stream Data and study the ways to assess Linked Stream Data processing engines. This deliverable uses the result of the work on Linked Stream Data and evaluates the $\text{SPARQL}_{\text{Stream}}$, both approaches were presented in the previous deliverable D1.1. The evaluation of SRBench on $\text{SPARQL}_{\text{Stream}}$ has triggered the collaboration between UPM and CWI to integrate $\text{SPARQL}_{\text{Stream}}$ with MonetDB/DataCell, which pertain to the topic of the following deliverable 1.7.

Document Information

IST Project Number	FP7 - 257641	**Acronym**	PlanetData
Full Title	PlanetData		
Project URL	http://www.planet-data.eu/		
Document URL	http://planet-data.eu/sites/default/files/pr-material/deliverables/D1.2_Benchmarking_RDF_storage_engines.pdf		
EU Project Officer	Leonhard Maqua		

Deliverable	**Number**	D1.2	**Title**	Benchmarking RDF storage engines
Work Package	**Number**	WP1	**Title**	Data Streams and Dynamicity

Date of Delivery	**Contractual**	M18	**Actual**	M18
Status	version 1.0		final ■	
Nature	prototype □ report ■ dissemination □			
Dissemination level	public ■ consortium □			

Authors (Partner)				
Responsible Author	**Name**	Ying Zhang	**E-mail**	Ying.Zhang@cwi.nl
	Partner	CWI	**Phone**	+31(0)20 592 4134

Abstract (for dissemination)	
Keywords	SRBench, Streaming RDF, benchmark, Linked Sensor Data

Version Log			
Issue Date	**Rev. No.**	**Author**	**Change**
2011-07-05	0.1	Ying Zhang, Oscar Corcho, Zoltán Miklós	Initial draft of TOC
2011-09-15	0.2	Ying Zhang, Pham Minh Duc, Peter Boncz	TOC revised
2011-09-20	0.3	Ying Zhang	Deliverable synopsis
2012-03-01	0.9	Ying Zhang, Pham Minh Duc, Fabian Groffen, Erietta Liarou, Peter Boncz, Martin Kersten, Jean Paul Calbimonte	Initial draft version for internal review
2012-03-22	0.95	Ying Zhang, Oscar Corcho, Pham Minh Duc	Revised version (for final assessment)
2012-03-30	1.0	Ying Zhang	Final version

Table of Contents

List of Figures and Tables

Abbreviations

DBMS	Database Management System
DSMS	Data Stream Management System
FOAF	Friend Of A Friend
LOD	Linked Open Data
LSD	Linked Stream Data
OWL	Web Ontology Language
QoS	Quality of Service
RDBMS	Relational Database Management System
RDF	Resource Description Framework
SSW	Semantic Sensor Web

1 Introduction

1.1 Background

Generally speaking, *data streams* are streams of data *continuously* generated at some regular or irregular *time intervals*. A data stream contains an infinite sequence S of $<v_i, t_i>$ pairs, where v_i is the data value and t_i is the timestamp at which v_i is valid. The values of t_i in the sequence S are monotonically non-decreasing, i.e., $i < j \rightarrow t_i \leq t_j$. Theoretically, v_i can be any value. Thus in case of streaming RDF data, we define an *RDF data stream* to be an infinite sequence S_{rdf} of $<spo_i, t_i>$ pairs, where spo_i is an RDF triple $<s_i, p_i, o_i>$ and t_i is the valid timestamp of the RDF spo_i.

The data that are dealt with by the data management systems (DBMS) and the most Semantic Web technologies are static data, which are characterised by the facts that the complete datasets are usually available beforehand and changes that will be made to the data are either in small amounts or at low frequencies, e.g., once in a day. The streaming data, on the contrary, are highly dynamic. There is no pre-collected dataset; instead, streaming data arrive on the fly, *continuously* at a *high rate*, e.g., once per second or even higher. Additionally, some data streams have only a *limited lifetime*. These characteristics of streaming data impose the real-time or near real-time requirement on the systems that handle this type of data, that is, such systems must carry out all work on a data item before the next data item arrives and within the lifetime of the data item.

Recent development in mobile technologies and wireless communication has resulted in an avalanche of streaming data. In just about a decade, streaming data has become ubiquitous in our daily life. Among others[1], sensor data is a major class of streaming data with the longest history. So, in the work presented in this document, we focus on this type of streaming data. Nowadays, sensors[2] have been adopted by a broad scope of applications, such as weather forecasting[3], traffic management[4], satellite imaging for earth observation[5], elderly care (Ruyter and Pelgrim, 2007) and seismic events detection[6]. Millions of sensors bring us not only a vast amount of data, but also data sources of various content, formats, modality and quality. This gives plenty of opportunities for new kinds of applications that utilise many data sources simultaneously, thus achieving functions not possible by using any single sensor network. Such applications require the use of heterogeneous and rapidly changing data sources in an integrated manner.

The data stream management systems (DSMSs) have focused on efficient managing and processing of streaming data (Chen et al., 2000; Carney et al., 2002; Abadi et al., 2003a; Abadi et al., 2003b; Babcock et al., 2004; Balakrishnan et al., 2004; Madden et al., 2002; Chandrasekaran and Franklin, 2002; Cranor et al., 2003; Aberer et al., 2006a; Aberer et al., 2006b; Ali et al., 2009; Gedik et al., 2008; Liarou et al., 2009; Franklin et al., 2009; Franklin et al., 2009; Chen and Hsu, 2010). These issues are mainly addressed in the context of individual sensor networks. The existing DSMSs do not provide tools to publish and share the streaming data; neither do they try to derive knowledge from the streaming data. As a result, applications dealing with streaming data are tied closely to one or a few sensor networks and are mostly only available within the same organisation. Semantic Web technologies, on the other hand, have focused on how to publish and interlink data on the World Wide Web, and how to perform complex reasoning tasks on the data. However, these technologies do not take into account rapidly changes of the data.

The lack of integration and communication between different sensor networks often *isolates* important data streams and intensifies the existing problem of *"too much (streaming) data but not enough knowledge"* (Sheth et al., 2008). The amount of streaming data has been growing extremely fast in the past several years and is expected to grow even faster in the coming decades. However, existing techniques are not able to gain

[1] Next to sensor data streams, there are also video streams and text streams.

[2] Throughout this document, we will use the word "sensors" to refer to both sensors and sensors networks.

[3] See: http://www.aemet.es

[4] See: http://www.tomtom.com/livetraffic/

[5] See: http://www.earthobservatory.eu/

[6] See: http://www.orfeus-eu.org/

all information from the available streaming data, letting alone interlinking streaming data with other (static) dataset to derive implicit information from the streaming data (Balazinska et al., 2007; Della Valle et al., 2009). To tackle this problem, researchers have set out for solutions in which Semantic Web technologies are adapted and extended for the publishing, sharing, analysing and understanding of the streaming data. Various proposals have emerged that address open issues such as how to apply reasoning on streaming data (Anicic et al., 2011; Della Valle et al., 2009; Sheth et al., 2008; Walavalkar et al., 2008; Whitehouse et al., 2006); how to publish raw streaming data in Semantic Web and connect them to the existing datasets on the Semantic Web (Bouillet et al., 2007; Corcho and García-Castro 2010; Le-Phuoc et al., 2009; Sequeda and Corcho 2009; Sheth et al., 2008); and how to apply the query language of semantic data on streaming data (Barbieri et al., 2009; Barbieri et al., 2010a; Barbieri et al., 2010b; Bolles et al., 2008; Calbimonte et al., 2010; Groppe et al., 2007; Groppe 2011; Hoeksema 2011; Le-Phuoc et al., 2010; Le-Phuoc et al., 2011a; Le-Phuoc et al., 2011b).

Among others, Sheth et al. (2008) first envision a *Semantic Sensor Web (SSW)*, in which sensor data is annotated with semantic metadata to increase interoperability as well as provide contextual information essential for situational knowledge[7]. Subsequently, Corcho and García-Castro (2010) identify the five most relevant challenges of the current Semantic Sensor Web. Della Valle et al. (2009) propose a novel approach, called *Stream reasoning*, that can provide the abstractions, foundations, methods and tools required to integrate data stream, the Semantic Web and reasoning systems. Sequeda and Corcho (2009) introduce the concept of *Linked Stream Data (LSD)*, a way in which the Linked Data principles can be applied to stream data so that stream data can be published as part of the Web of Linked Data. These visions are answered by a number of streaming RDF processing engines that, in general, extend the RDF data model with a notion of time and the W3C standard RDF query language, SPARQL, with features to express continues queries (Groppe et al., 2007; Groppe 2011; Bolles et al., 2008; Barbieri et al., 2009; Barbieri et al., 2010a; Barbieri et al., 2010b; Le-Phuoc et al., 2009; Le-Phuoc et al., 2010; Le-Phuoc et al., 2011a; Le-Phuoc et al., 2011b; Calbimonte et al., 2010; Anicic et al., 2011; Hoeksema, 2011). The increasing interest in streaming RDF processing engines calls for a standard way to compare the different systems.

1.2 RDF/SPARQL

The *Resource Description Framework (RDF)* is a family of World Wide Web Consortium (W3C)[8] specifications originally designed as a metadata data model. It has come to be used as a general method for conceptual description or modeling of information that is implemented in web resources using a variety of syntax formats[9]. RDF extends the linking structure of the Web to use URIs for naming relationships between things as well as the resources that are related (usually referred to as a "*triple*"). Using this simple model allows structured and semi-structured data to be mixed, exposed and shared across different applications. This linking structure forms a directed, labeled graph where the edges represent the named link between two resources represented by the graph nodes. This graph view is the easiest possible mental model for RDF and is often used in easy-to-understand visual explanations[10].

The *SPARQL Protocol and RDF Query Language (SPARQL)* is an RDF query language, i.e., a query language for databases that is able to retrieve and manipulate data stored in RDF format (Rapoza, 2006; Segaran et al., 2009). It was made a standard by the *RDF Data Access Working Group (DAWG)*[11] of the World Wide Web Consortium, and considered as one of the key technologies of semantic web (Rapoza, 2006). On 15 January 2008, SPARQL 1.0 became an official W3C Recommendation[12]. SPARQL[13] allows

[7] Situational knowledge is the knowledge specific to a particular situation.

[8] http://www.w3.org/

[9] http://en.wikipedia.org/wiki/Resource_Description_Framework

[10] http://www.w3.org/RDF/

[11] http://www.w3.org/2009/sparql/wiki/Main_Page

[12] http://www.w3.org/blog/SW/2008/01/15/sparql_is_a_recommendation/

[13] http://www.w3.org/TR/rdf-sparql-query/

for a query to consist of triple patterns, conjunctions, disjunctions, and optional patterns. The syntax of SPARQL is similar to the SQL language for querying RDF graphs[14].

1.3 SRBench

Following a long tradition in database research, in this deliverable we present a benchmark, called *SRBench*, to assess the abilities of a streaming RDF engine to cope with a broad range of different query types typically encountered in real-world scenarios. This benchmark can help both researchers and users to compare streaming RDF engines in a standardized application scenario.

To the best of our knowledge, SRBench is the first general purpose benchmark that has been primarily designed to assess the streaming RDF storage engines. In general, little work has been done on benchmarks for streaming data management systems. The Linear Road Benchmark (Arasu et al., 2004) is the only publicly available streaming benchmark, but unfortunately, it is not ideal to assess streaming RDF engines. Since the benchmark was primarily designed to assess traditional data stream management systems, which are based on the relational data model, it does not capture the properties of RDF data, which use the graph data model. Moreover, the benchmark does not take into account interlinking the benchmark data with other datasets; neither does it consider reasoning in its queries. Although several RDF/SPARQL benchmark exist (Guo et al., 2005; Bizer and Schultz 2009; Schmidt et al., 2009), they do not capture the properties of streaming data. In Le-Phuoc et al. (2011b) and Hoeksema (2011), some micro benchmark queries are created for preliminary evaluation of the proposed systems. However, the queries were designed with a particular system in mind and they only cover a small subset of the features of RDF/SPARQL. Moreover, simply streaming parts of the data of an existing benchmark does not create a realistic scenario (Hoeksema, 2011). Hence, they cannot serve as general-purpose benchmarks.

Since streaming RDF processing is still a developing research area, the *first goal* of SRBench is to evaluate the functional completeness of a streaming RDF engine. To this end, we offer a set of queries where each query is intended to challenge a particular aspect of the query processor. The benchmark contains a concise, yet comprehensive set of queries which covers the major aspects of streaming SPARQL query processing, ranging from simple pattern matching queries to queries with complex reasoning tasks. The main advantages of applying Semantic Web technologies on streaming data include providing better search facilities by adding semantics to the data, reasoning through ontologies, and integration with other data sets. The ability of a streaming RDF engine to process these distinctive features is accessed by the benchmark with queries that apply reasoning not only over the streaming sensor data, but also over the metadata and even other data sets in the Linked Open Data (LOD) cloud, currently including the RDF version of the GeoNames dataset[15] and the DBpedia dataset[16].

Duan et al. (2011) present an extensive study of the characteristics of real-world RDF data and generated (i.e., synthetic) benchmark RDF data. The study shows that existing benchmarks do not accurately predict the behaviour of RDF stores in realistic scenarios. For this reason, all datasets used in SRBench are real-world datasets, including the LinkedSensorData dataset[17], the RDF version of the GeoNames dataset[15] and the DBpedia dataset[16]. The choices for the three datasets were made on purpose. The LinkedSensorData dataset publishes real-world sensor metadata and sensor observation data according to the Linked Stream Data principle, which was studied in the previous PlanetData deliverable D1.1 (Corcho et al., 2011). Moreover, LinkedSensorData is the largest sensor dataset in both the Linked Open Data[18] cloud and the CKAN[19] data portal. The LinkedSensorData dataset links the sensor locations to nearby geographic places defined by the GeoNames dataset, so this naturally determines our choice of the GeoNames dataset. The choice for the DBpedia dataset is also a matter of course, since DBpedia is the largest and the most popularly

[14] http://www.thefigtrees.net/lee/sw/sparql-faq#what-is

[15] See: http://www.geonames.org/ontology/documentation.html

[16] See: http://wiki.dbpedia.org/Downloads37

[17] See: http://wiki.knoesis.org/index.php/LinkedSensorData

[18] See: http://richard.cyganiak.de/2007/10/lod/

[19] See: http://thedatahub.org/dataset/knoesis-linked-sensor-data

used dataset in the Linked Open Data cloud. Moreover, the PlanetData partner FUB is one of the originators of DBpedia.

Streaming RDF processing is an evolving topic. As a result of this, the proposed systems are in their earlier stages of development. At this moment, one of the most important issues is to assess the functionality of the proposed systems. Do they provide a sufficient set of functions that are needed by the streaming applications? Do they miss any crucial functionalities? Do they provide any additional functionalities that can be beneficial for streaming applications, which thus distinguish themselves from other systems in the same area. Therefore, we complement our work on SRBench with a functional evaluation of the benchmark on the $\text{SPARQL}_{\text{Stream}}$ query language and processing engine[20] developed by the PlanetData partner UPM (Calbimonte et al., 2010). These results are intended to give a first baseline and illustrate the state of the art. Although the works proposed by (Barbieri et al., 2010a; Barbieri et al., 2010b) and by (Le-Phuoc et al., 2011a; Le-Phuoc et al., 2011b) are competitive with $\text{SPARQL}_{\text{Stream}}$, we find $\text{SPARQL}_{\text{Stream}}$ the best starting point for the evaluation of the benchmark for several reasons. Among all streaming SPARQL extensions, the syntax of the $\text{SPARQL}_{\text{Stream}}$ extension is the *cleanest*, because it only extends the `FROM` clause to also allow streaming resources. $\text{SPARQL}_{\text{Stream}}$ is the *only* extension that is based on both the newest W3C standard SPARQL 1.1 language (Harris and Seaborne, 2012) and CQL (Arasu et al., 2003b), the best-defined continuous query language for DSMSs. Moreover, the $\text{SPARQL}_{\text{Stream}}$ engine is implemented in extensible modules, which makes it easy to be integrated with MonetDB/DataCell, a continuous query processor being developed by the PlanetData partner CWI, which will be part of the main objective of the following deliverable D1.7 of PlanetData.

1.4 Outline

This document is further organised as follows. In Section 2, we give an extensive study of the state-of-the-art techniques in the related research areas, which include Data Streams Management Systems (DSMSs), streaming benchmarks, streaming RDF processing engines, SPARQL language extensions for continues queries, and RDF/SPARQL benchmarks. In Section 3, we give some brief background information about the datasets which are used in the SRBench benchmark. In Section 4, we specify the data model of SRBench. In Section 5, we define the benchmark queries. In Section 6, we give a functional evaluation of the benchmark using the $\text{SPARQL}_{\text{Stream}}$ query processing engine developed by the PlanetData partner UPM. Finally, in Section 7, we conclude and discuss future work.

[20] See: http://code.google.com/p/semanticstreams/source/checkout

2 Related Work

Before starting with designing a new benchmark for streaming RDF engines, we have first carried out an extensive study of the state-of-the-art of the related techniques in the relational world as well as the RDF/SPARQL world. In this section, we highlight several notable existing work in the areas of Data Streams Management Systems (DSMSs), streaming benchmarks, streaming RDF processing engines, SPARQL language extensions for continues queries, and RDF/SPARQL benchmarks.

2.1 Data Stream Management Systems

Modern applications coming from various fields (e.g., finance, telecommunications, networking, sensor and web applications) require fast data analysis over data that are continuously updated. In this new type of applications, called *data stream applications*, we first of all need mechanisms to support *long-standing/continuous* queries over data that is continuously, and at high rate, updated by the environment. To achieve good processing performance, i.e., handling input data within strict time bounds, a system should provide *incremental* processing where query results are frequently and instantly updated as new data arrives. Systems should scale to handle numerous co-existing queries at a time and exploit potential similarities between the large number of standing queries. Furthermore, environment and workload changes may call for *adaptive* processing strategies to achieve the best query response time. Even if conventional DBMSs are powerful data management systems, the hooks for building a continuous streaming application are not commonly available in such systems.

Given these differences, and the unique characteristics and needs of continuous query processing, the pioneering Data Stream Management Systems (DSMS) architects naturally considered that the existing DBMS architectures were inadequate to achieve the desired performance. Another aspect is that the initial stream applications had quite simple requirements in terms of query processing. This made the existing DBMS systems look overloaded with functionalities. These factors led researchers to design and build new architectures from scratch. Several DSMS solutions have been proposed over the last years giving birth to very interesting ideas and system architectures. In this section, we discuss several characteristic DSMSs research prototypes.

2.1.1 Aurora

Aurora (Carney et al., 2002; Abadi et al., 2003a; Abadi et al., 2003b; Babcock et al., 2004; Balakrishnan et al., 2004) is a data stream management system that was developed between 2001 and 2004, as a result from the collaboration of three research groups from MIT, Brown University and Brandeis University.

Aurora uses the *boxes and arrows* paradigm, followed in most workflow systems. Each *box* represents a query operator and each *arc* represents a data flow or a queue between the operators. Each query is built out of a set of operators and all submitted queries constitute the Aurora query network. *SQuAl* is Aurora's query algebra that provides nine stream-oriented primitive operators, i.e., `Filter`, `Map`, `Union`, `Aggregate`, `Join`, `BSort`, `Resample`, `Read`, and `Update`. The users can construct their queries out of these operators. Each operator may have multiple input streams (e.g., `union`), and could give its output to multiple boxes (e.g., `split`). Tuples flow through an acyclic, directed graph of processing operators. At the end, each query concludes to a single output stream, presented to the corresponding application. Aurora can also maintain historical storage, to support ad hoc queries.

The query network is divided into a collection of *n* sub-networks. The application administrator makes the decision where to insert the connection points. Connection points indicate the network modification points and specify the query optimization limits. New boxes can only be added to or deleted from the connection network points over time. Instead of trying to optimize the whole query graph at once, the Aurora optimizer optimises it piece-by-piece. It isolates each sub-network, surrounded by connection points, individually from the rest of the network and optimises it in a periodic manner.

Figure 1 illustrates the high-level system model of the Aurora system, as presented by the authors in their original publications (Carney et al., 2002). The *router* connects the system to the outside world. It receives input data streams from the external data sources, e.g., sensors, and from inside boxes. If the query processing is completed, the router forwards the tuples to external waiting sources; otherwise it re-feeds them to the storage manager for further processing. The *storage manager* stores and retrieves the data

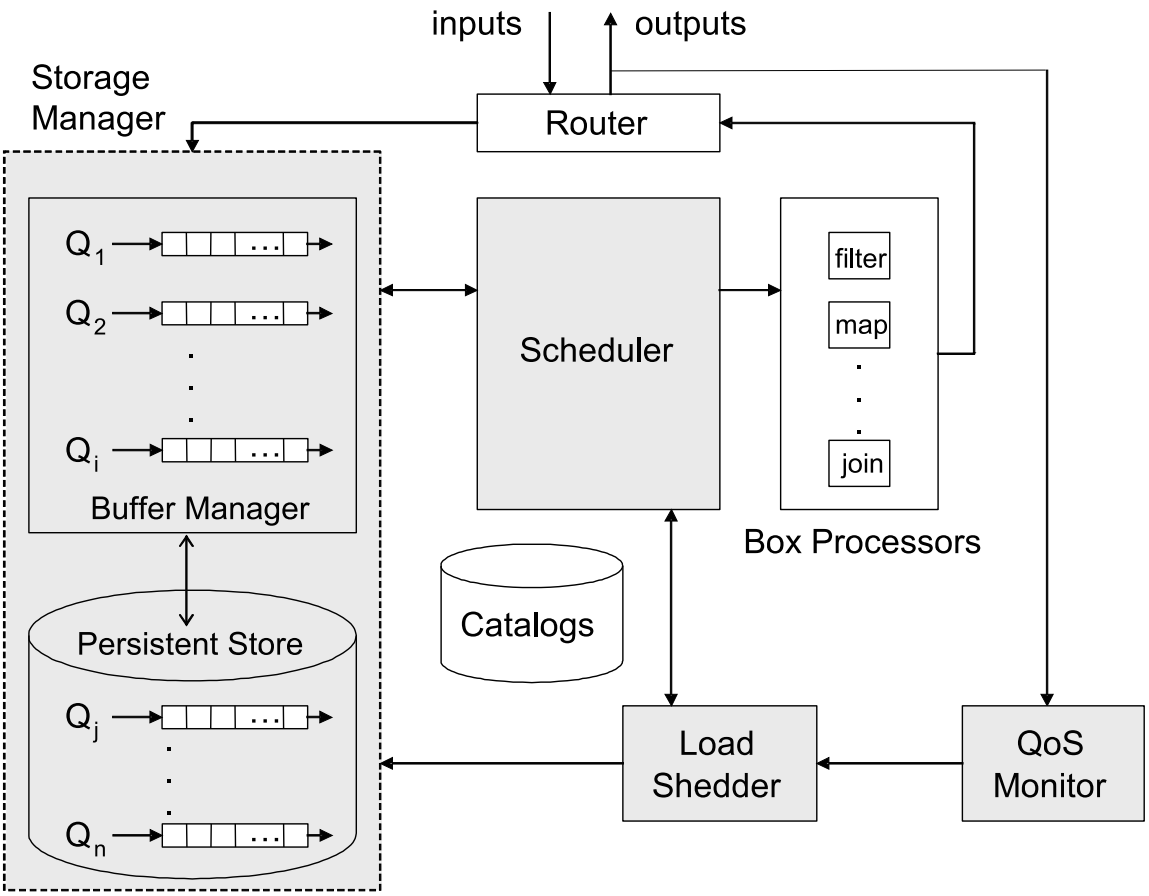

Figure 1: Aurora System Architecture (Carney et al., 2002)

streams on in-memory buffers between query operators. Also, it maintains historical storage, to serve potential ad-hoc queries. A persistence specification indicates exactly for how long the data is kept.

The *scheduler* is the core Aurora component (Babcock et al., 2004). It decides when an operator should be executed, and feeds it with the appropriate number of queued tuples. In Aurora, there is one *box processor* per operator type; this part is responsible for executing a particular operator when the scheduler calls it. Then, the box operator forwards the output tuples to the router. The scheduler continuously monitors the state of the operators and the buffers and repeats this procedure periodically.

The designers of Aurora dedicated a big part of their research on addressing methods that guarantee Quality of Service (QoS) requirements when the system becomes overloaded (Tatbul, 2007). They proposed load-shedding techniques that attach to the query network a type of system-level operators that selectively drop tuples. Aurora applies such operators when the rate on incoming streams overwhelm the stream engine, trying to balance between the expected side-effect on result accuracy, while meeting QoS application requirements. Later on, Medusa (Sbz et al., 2003) and Borealis (Abadi et al., 2005) extended the single-site Aurora architecture to a distributed setting. In 2003, the original research prototype was commercialized into a start-up company named StreamBase Systems[21].

2.1.2 STREAM

The *STanford stREam datA Management (STREAM)* system (Motwani et al., 2002; Arasu et al., 2003a) is another data stream processing research prototype that was designed and developed at Stanford University from 2001 to 2006.

STREAM provides a declarative query language, called *CQL* (Arasu et al., 2003b), which allows queries to handle data from continuous data streams as well as conventional relations. CQL extends SQL by allowing stream and relational expressions, and introducing window operators. In CQL there are three classes of

[21] StreamBase Systems, Inc (2003). See: http://www.streambase.com/

operators: (a) the *stream-to-relation* operators, which produce a relation from a stream (i.e., sliding windows), (b) the *relation-to-relation* operators, which produce a relation from one or more other relations, such as in relational algebra and SQL and (c) the *relation-to-stream* operators, i.e., `Istream`, `Dstream`, and `Rstream`, which produce a stream from a relation. There are also three classes of sliding window operators, i.e., time-based, tuple-based and partitioned. However, in practice it does not support sliding windows with a slide bigger than a single tuple.

In STREAM, operators read from and write to a single or multiple *queues*. Furthermore, synopses are attached to operators and store their intermediate state. This is useful when a given operator needs to continue its evaluation over an already processed input, for instance, the content of a sliding window or the relation produced by a subquery. Synopses are also used to summarize a stream or a relation when approximate query processing is required. Scheduling in STREAM also happens at the operator level as it used to in stream systems. It uses either a simple scheduling strategy (Motwani et al., 2003) such as round robin or FIFO, or the more sophisticated Chain algorithm (Babcock et al., 2003). The scheduling methods in STREAM focus on providing run-time memory minimisation. STREAM also includes a monitoring and adaptive query processing infrastructure called *StreaMon* (Babu and Widom, 2004), which consists of three components. The *Executor* runs query plans and produces results. The *Profiler* collects statistics about stream and query plan characteristics. Finally, the *Reoptimizer* takes the appropriate actions to always ensure that the query plan and memory usage are optimal for the current input characteristics. Whenever not enough CPU or memory is available, the system proceeds with approximate query processing, trying to handle the query load by sacrificing accuracy. StreaMon introduces random sampling operators into all query plans, in a way that the relative error is the same for all queries. STREAM deals with memory-limitations also by discarding older tuples from the window joins operators, leaving free space for new data. The goal here is to maximize the size of the resulting subset.

2.1.3 Telegraph-CQ

TelegraphCQ (Chandrasekaran et al., 2003) is a continuous query processing system built at University of California, Berkeley. The main focus is on adaptive and shared continuous query processing over query and data streams. The team in Berkeley built TelegraphCQ based on previous experience obtained while developing the preliminary prototypes *CACQ* (Madden et al., 2002) and *PSoup* (Chandrasekaran and Franklin, 2002).

PSoup addresses the need for treating data and queries symmetrically. Thus, it allows new queries to see old data and new data to see old queries. This feature is passed to the TelegraphCQ architecture as well. Furthermore, TelegraphCQ successfully addresses and resolves important limitations that were not addressed in previous prototypes, e.g., it deals with memory and resource limitations, while trying to guarantee QoS over acceptable levels; and it focuses on scheduling and resource management of groups of queries. TelegraphCQ constructs query plans with adaptive routing modules, called *Eddies* (Avnur and Hellerstein, 2000). Thus, it is able to proceed to continuous run-time optimizations, dynamically adapting to the workload. The Eddies modules adaptively decide how to route data to appropriate query operators on a tuple-by-tuple basis.

TelegraphCQ tries to leverage the infrastructure of a conventional DBMS, by reusing a big part of the open source PostgreSQL[22] code base. With minimal changes at particular components, it tries to use the front-end piece of code that PostgreSQL already offers, including the Postmaster, the Listener, the System Catalog, the Query Parser and the PostgreSQL Optimiser. However, the TelegraphCQ developers proceeded to significant changes on the deeper PostgreSQL parts, such as the Executor, the Buffer Manager and the Access Methods, to make them compatible with the unique requirements of stream processing. Figure 2 illustrates an overview of the TelegraphCQ architecture as it is originally presented in (Chandrasekaran et al., 2003). The rightmost oval part is the most solid contribution of PostgreSQL to the new system architecture. The processes included in there, are connected using a shared memory infrastructure, and the generated query plans are placed in a query plan queue. From there, the *Executor* picks them up to proceed with the actual processing, trying first to classify the plans into groups for sharing work. The query results are continuously placed in the output queues. The *Wrapper* mechanism allows data to be streamed into the system.

[22] See: http://www.postgresql.org/

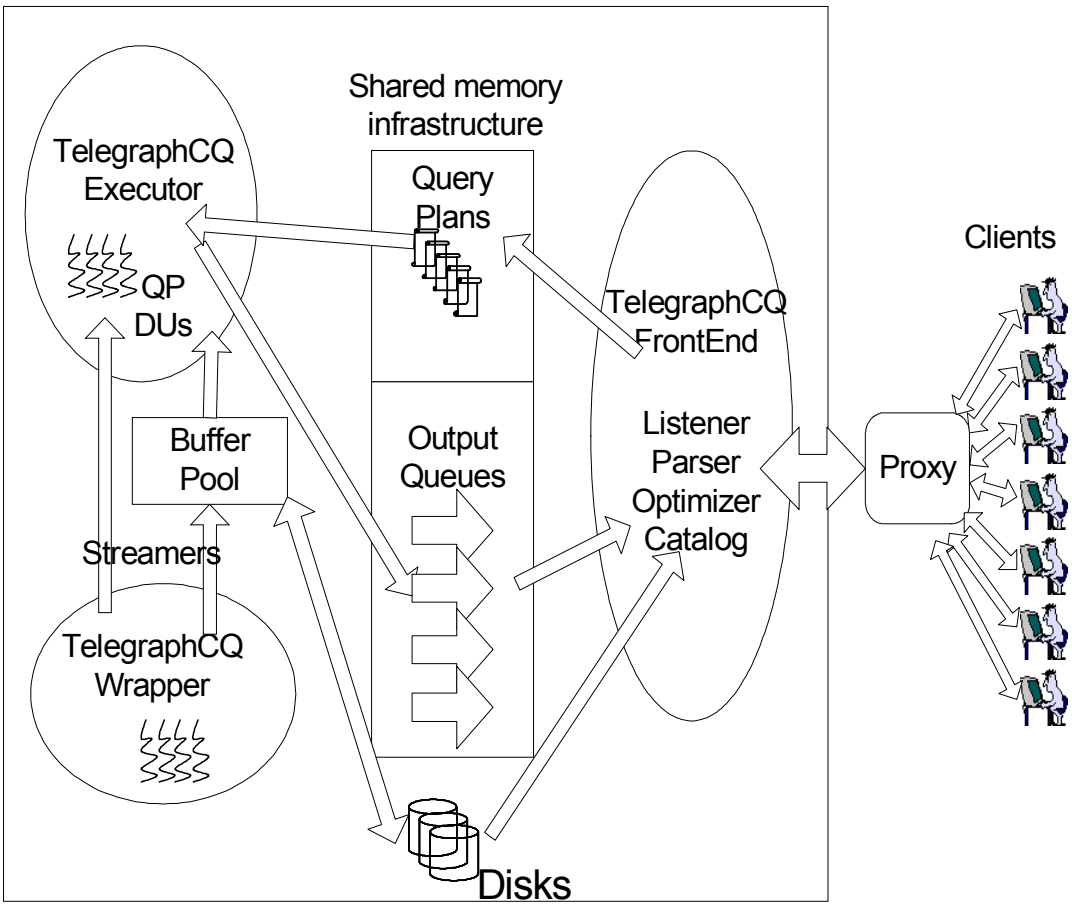

Figure 2: TelegraphCQ System Architecture (Chandrasekaran et al., 2003)

2.1.4 Other DSMSs

The unique requirements of monitoring applications, establish a new research field that demonstrates interesting results on new system architectures, query languages, specialised algorithms and optimisations. So far, we presented three characteristic efforts from the academic world. However the research efforts do not stop there. Many other interesting stream systems have been presented to related journals and conferences. Some of them have been matured into commercial products.

A noteworthy result is *Gigascope* (Cranor et al., 2003), a lightweight stream processing system that was developed in AT&T to serve network applications. It emerged from requirements of the company itself, e.g., traffic analysis, network monitoring and debugging.

NiagaraCQ (Chen et al., 2000) is an XML-based continuous query system that focuses on query optimisation to improve scalability. This system tries to exploit query similarities to group queries and potentially save processing cost. The grouping process happens incrementally and once new queries are added to the system, they find their place in the appropriate groups.

Different language semantics are introduced in the *Cayuage* system, developed in Cornell University. Cayuage is a stateful publish/subscribe system based on a non-deterministic finite state automata (NFA) model.

Big vendors like Microsoft (Ali et al., 2009), IBM (Gedik et al., 2008) and Aleri/Coral8[23] have also become active in the data stream area during the last few years, developing high performance complex event processing systems. Their focus is on pure stream processing, providing additional external access to historical data. Furthermore, they have moved their architectures in distributed settings to cope with the increasing data requirements.

[23] See: http://www.coral8.com/

2.1.5 A New Processing Paradigm: Persistent and Stream Data Under the Same Roof

In the previous section, we discussed the main philosophy of the specialized stream engines that were developed to efficiently handle continuous query processing in bursty data arrival periods. However, the technological evolutions keep challenging the existing architectures with new application scenarios. In recent years, a new processing paradigm is born (Liarou et al., 2009; Chen and Hsu, 2010; Franklin et al., 2009) where incoming data needs to quickly be analysed and possibly be combined with existing data to discover trends and patterns. Subsequently, the new data enters the data warehouse and is stored for further analysis if necessary. This new paradigm requires scalable query processing that combines continuous and conventional processing.

The *Large Synoptic Survey Telescope (LSST)*[24] is a grandiose paradigm. In 2018 the astronomers will be able to scan the sky from a mountaintop in Chile, recording 30 Terabytes of data every night, which will incrementally lead a 150 Petabyte database over the operation period of ten years[25]. It will be capturing changes to the observable universe evaluating huge statistical calculations over the entire database.

The *Large Hadron Collider (LHC)*[26] is another characteristic data-driven example. It is a particle accelerator that is expected to revolutionise our understanding for the universe, generating 60 Terabytes of data every day (4Gb/sec). The same model stands for modern data warehouses which enrich their data on a daily basis creating a strong need for quick reaction and combination of scalable stream and traditional processing (Winter and Kostamaa, 2010). However, neither pure database technology nor pure stream technology are designed for this purpose.

The *Truviso Continuous Analytics system* (Franklin et al., 2009), a commercial product of Truviso, is another recent example that follows a similar approach as TelegraphCQ. Part of the team that has worked on the TelegraphCQ project proceeded to the commercialised version of the original prototype. They extend the open source PostgreSQL[22] database to enable continuous analysis of streaming data, tackling the problem of low latency query evaluation over massive data volumes. *TruCQ* integrates streaming and traditional relational query processing in such a way that it ends-up to a *stream-relational* database architecture. It is able to run SQL queries continuously and incrementally over data while they are still coming and before they are stored in *active database tables*. TruCQ's query processing outperforms traditional store-first-query-later database technologies as the query evaluation has already started when the first tuples arrive. It allows evaluation of one-time and continuous queries as well as combinations of both query types.

Another recent work, coming from the HP Labs (Chen and Hsu, 2010), confirms the strong research attraction for this trend. They define an extended SQL query model that unifies queries over both static relations and dynamic streaming data, by developing techniques to generalise the query engine. They extend the PostgreSQL[22] database kernel, and build an engine that can process persistent and streaming data in a single design. First, they convert a stream into a sequence of *chunks* and then continuously call the query over each sequential chunk. The query instance never shuts down between the chunks, which thus forms a cycle-based transaction model.

2.1.6 Data Stream Query Languages

The unique monitoring application requirements determined the development of new data management architectures and consequently the need for new querying paradigms. In the literature we distinguish two classes for query languages that define the proper data streaming semantics.

Declarative

Many stream systems define and support languages that maintain the declarative and rich expressive power of SQL. A characteristic example is *CQL* (Continuous Query Language) (Arasu et al., 2003b), which is introduced and implemented in the STREAM prototype (Motwani et al., 2002; Arasu et al., 2003a). Apart from streams, CQL also includes relations. Thus, we can write queries from each category as well as queries combining both data types. In CQL, there are three types of operators: the relation-to-relation operators that

[24] See: http://www.lsst.org/

[25] See: http://www.lsst.org/lsst/public/tour_software

[26] See: http://lhc.web.cern.ch/lhc/

SQL already offers; the stream-to-relation operators that reflect the sliding windows; and the relation-to-streams operators that produce a stream from a relation. There are also three classes of sliding window operators in CQL: time-based, tuple-based and partitioned windows. One can denote a time-based sliding window of size *T* on a stream *S*, with the expression `[Range T]`. Additionally, one can specify a tuple-based sliding window of size *N* on a stream *S* by following the reference to *S* in the query with `[Rows N]`.

GSQL is another SQL-like query language, developed for Gigascope to express queries for network monitoring application scenarios. GSQL is a *stream-only* language, thus all inputs to a GSQL operator should be streams and the outputs are streams as well. However, relations can be created and manipulated using user-defined functions. Each stream should have an ordering attribute, e.g., a timestamp. Gigascope only supports a subset of the operators in SQL, i.e., selections, aggregations and joins of two streams. In addition to these operators, GSQL includes a stream *merge* operator that works as an order-preserving union of ordered streams. In GSQL, only landmark windows are supported directly, but sliding windows may be simulated via user-defined functions.

StreaQuel is the declarative query language proposed and used in TelegraphCQ prototype. It supports continuous queries over a combination of tables and data streams. By using a *for-loop* construct with a variable *t* that moves over the timeline as the for-loop iterates, one can express the sequence of windows over which the user desires the answers to the query. Inside the loop one can include a *WindowIs* statement that specifies the type and size of the window over each stream. This way, snapshot, landmark and sliding window queries can be easily expressed.

Procedural

A different approach to the declarative SQL-like query languages is a procedural one. For instance in Aurora, the developers proposed *SQuAl* (Stream Query Algebra), a boxes-and-arrows query language. Through a graphical interface, a user can draw a query plan by placing boxes (i.e., operators) and arrows (i.e., data streams) in the appropriate order, and specifying how the data should flow through the system. SQuAl accepts streams as inputs and returns streams as output. However, it gives the option to the user to include historical data to query processing through explicitly defined connection points.

2.1.7 Discussion

In this section, we presented some well known data stream management systems, such as Aurora, STREAM, TelegraphCQ, Gigascope, NiagaraCQ and Cayuage. Each one contributed in a unique way to the broad research area of data streams. In general, all of them follow the same philosophy, that is, they are built from scratch. Although they basically all use an SQL-based language, the query processing engines dismiss the conventional database technology. Furthermore, the existing DSMSs have mainly concentrated on efficient processing of binary data streams. None of them deal with the semantic aspects of the data.

2.2 Linear Road Benchmark

The Linear Road Benchmark (Arasu et al., 2004) is the only benchmark developed for evaluating traditional data stream engines.[27] It is a highly challenging and complicated benchmark due to the complexity of the many requirements. It stresses the system and tests various aspects of its functionality, e.g., window-based queries, aggregations, various kinds of complex join queries; theta joins, self-joins, etc. It also requires the ability to evaluate not only continuous queries on the stream data, but also historical queries on past data. The system should be able to store and later query intermediate results. Due to the complexity, only a handful of implementations of the benchmark exist so far. Most of them are based on a low level implementation in C that represents a specialized solution that does not clearly reflects the generic potential of a system. In this paper, we implemented the benchmark in a generic way using purely the DataCell model and SQL. We created numerous SQL queries that interact with each other via result forwarding (details are given below).

[27] Although commercial companies with streaming products have their own internal benchmarks, for instance, STAC, the Securities Technology Analysis Center (see: http://www.stacresearch.com/node/2), they are open only to subscriber companies and they don't publish their reports.

The benchmark simulates a traffic management scenario where multiple cars are moving on multiple lanes and on multiple different roads. The system is responsible to monitor the position of each car. It continuously calculates and reports to each car the tolls it needs to pay and whether there is an accident that might affect it. An accident is detected when two or more cars are in the same position for 4 continuous timestamps. In addition, the system needs to continuously monitor historical data, as it is accumulated, and report to each car the account balance and the daily expenditure. Furthermore, the benchmark poses strict time deadlines regarding the response times which must be up to X seconds, i.e., an answer must be created at most X seconds after all relevant input tuples have been created. X is 5 or 10 seconds depending on the query (details below).

The benchmark contains a tool that creates the data and verifies the results. The data of a single run reflects three hours of traffic, while there are multiple scale factors that increase the amount of data created for these three hours, e.g., for scale factor 0.5 the system needs to process $6 * 10^6$ tuples, while for scale factor 1 we need to process $1.2 * 10^7$.

2.3 Streaming RDF Engines

Triggered by the increasing needs for gaining more knowledge from the rapidly growing amount of streaming data (Balazinska et al., 2007; Sheth et al., 2008; Della Valle et al., 2009), a new research direction has recently emerged from the Semantic Web community. Several streaming RDF engines have been proposed, in which the Semantic Web researchers try to combine data stream management technologies and RDF/SPARQL technologies for the publishing, annotating and reasoning of streaming data. In this section, we highlight several notable proposals in this research area.

2.3.1 StreamSPARQL

Bolles et al. (2008) firstly extended the SPARQL 1.0 language (Prud'hommeaux and Seaborne, 2008) with window-based processing of RDF streams. At the syntax level, the *StreamSPARQL* language allows both time-based and element-based windows. Further in this proposal, the authors have focused on the work at the algebraic level. Data streams specific new operators, such as *sliding δ-window* and *sliding tuple window*, are added to the SPARQL 1.0 algebra. Existing SPARQL 1.0 algebraic operators, e.g., *Filter*, *Union* and *Join*, are extended to cope with RDF data streams. As a first proposal on streaming RDF processing, several choices made by the authors call for reconsideration.

First, essential ingredients for streaming data processing, such as aggregation and timestamp function, are omitted in (Bolles et al., 2008). With these limitations, the proposed language does not have the power to express many queries that are common in streaming applications. Furthermore, the authors do not follow the established approach in DSMSs where windows are used to transform streaming data into non-streaming data in order to apply standard algebraic operations. Instead, the authors have chosen to change the standard SPARQL operators by making them timestamp-aware, which effectively introduces new language semantics.

In StreamSPARQL, Windows can be defined not only in the `FROM` part of a SPARQL query to define a common window for the whole data stream, but also in the graph patterns to allow a finer granularity. On the one hand, this makes the query syntax more intricate, as window clauses can appear in multiple places. On the other hand, it complicates query evaluation. Since window operations are no longer required to be at the leaves of the query tree, they need to be interleaved with standard SPARQL operations, possibly interfering with the separation of concerns between stream management and query evaluation. Unfortunately, the authors did not pay special attention to the impact of this flexibility at the language level on the query processing; neither did they provide an engine to handle the proposed language.

2.3.2 C-SPARQL

The *Continuous SPARQL (C-SPARQL)* (Barbieri et al., 2009; Barbieri et al., 2010a; Barbieri et al., 2010b) is another proposal to extend the SPARQL 1.0 language for continuous queries over streams of RDF data. At the syntax level, C-SPARQL is inspired by continuous query languages for relational data streams, such as CQL (Arasu et al., 2003b). C-SPARQL extends SPARQL 1.0 with features including query registration, stream registration, and both time and triple based windows. C-SPARQL assumes that each data stream is associated with a distinct IRI, which is a locator of the actual data source of the stream. Having learnt from the limitations of StreamSPARQL, C-SPARQL has further extended the syntax of SPARQL 1.0 by

aggregates and timestamp functions, but restrict the functionalities by allowing streams to be only at the leaves of the query trees and only one Window per stream.

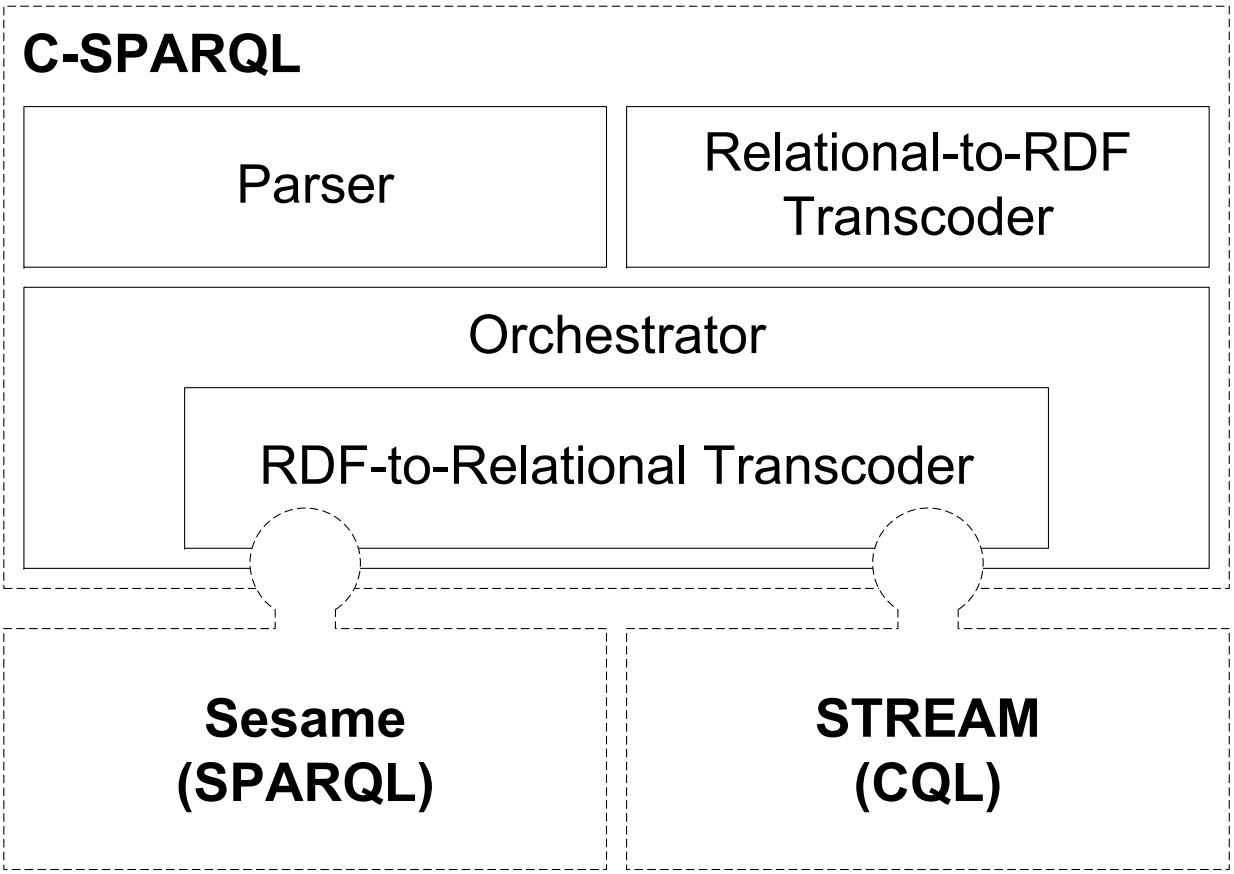

Figure 3: C-SPARQL Architecture Overview (Barbieri et al., 2010b)

Barbieri et al. (2010b) propose an execution framework for C-SPARQL based on a plug-in architecture, to leverage existing technologies in the areas of data streams management systems (DSDM) and SPARQL query processing engines. Figure 3 shows the architecture of the proposed framework as it is originally presented in (Barbieri et al., 2010b). The architecture relies entirely on existing technologies. The SPARQL reasoner plug-in is used to evaluate the static part of the query. An existing relational data stream management system is used to evaluate both streams and aggregates. A parser parses the C-SPARQL query and hands it over to the orchestrator, which subsequently translates the query in a static part, i.e., a SPARQL query, and a dynamic part, i.e., a CQL query. The SPARQL query is used to extract the static knowledge from the reasoner, while the CQL query is registered in the DSMS. The combination of static RDF data with streaming information yields possibilities for stream reasoning, an important step enabling reasoners to handle rapidly changing data in addition to static knowledge.

Among all streaming RDF engines, C-SPARQL is one of the more mature systems. Since the system has switched to support the SPARQL 1.1 language in a later version (Barbieri et al., 2010a), its interoperability and usefulness have also been largely improved.

2.3.3 SPARQL$_{Stream}$

Calbimonte et al. (2010) propose two extensions to existing work to enable *ontology-based access to streaming data sources*, e.g., sensor networks, through declarative continuous queries. In the proposed system, sources can link their data content to ontologies through *S_2O (Stream-to-Ontology)* mappings. Subsequently, users can query the ontology using the *SPARQL$_{Stream}$* query language.

S_2O is an extension of the *R_2O (Relational-to-Ontology)* language (Barrasa et al., 2004), which enables ontology-based access to relational data sources by defining a mapping from relational data sources to ontologies. S_2O extends R_2O with mappings from streaming sources to ontologies. SPARQL$_{Stream}$ is another continuous query language that extends SPARQL 1.1 to process streaming RDF data. It extends the `FROM`

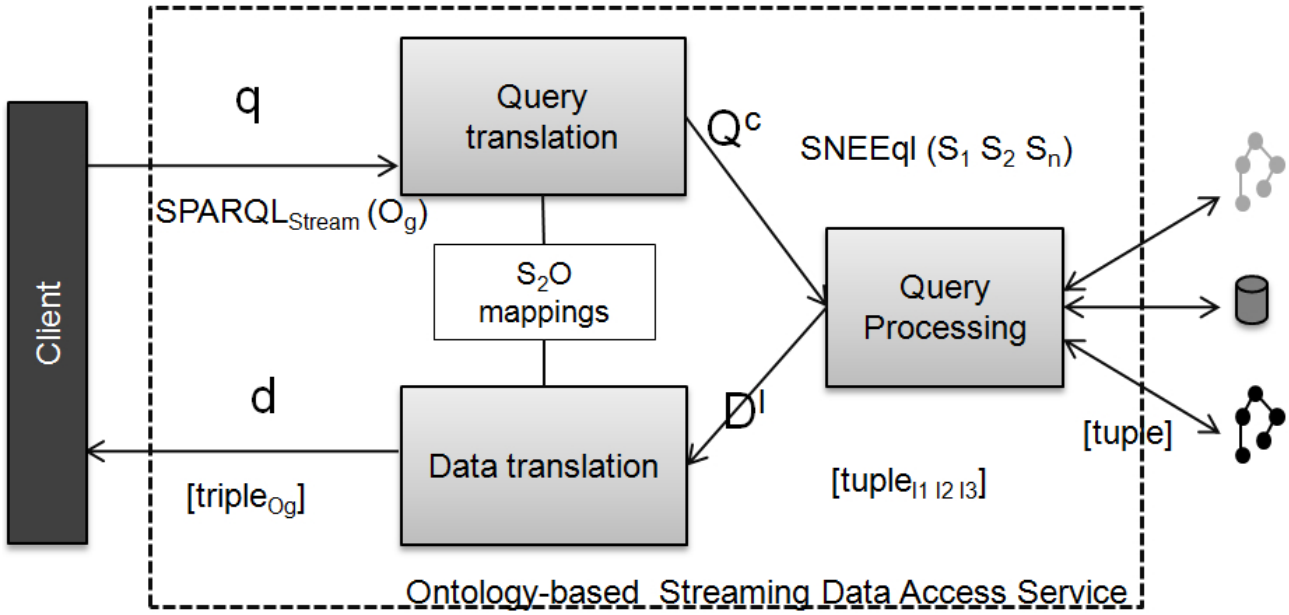

Figure 4: Ontology-based streaming data access service (Calbimonte et al., 2010)

clause with stream data sources and time-based windows on the streams. It allows time windows to be defined in the past so as to support correlation with historic data. Inspired by CQL (Arasu et al., 2003b) and SNEEql (Brenninkmeijer, et al., 2008), $SPARQL_{Stream}$ also supports three window-to-stream operators, i.e., `Istream`, `Dstream`, and `Rstream`, to produce a stream from the result of a window operator.

The overall architecture of the ontology-based streaming data access service is depicted in Figure 4. The service receives queries specified in terms of the classes and properties of the ontology using $SPARQL_{Stream}$. In order to transform the $SPARQL_{Stream}$ query, expressed in terms of the ontology, into queries in terms of the data sources, a set of mappings must be specified. These mappings are expressed in S_2O. This transformation process is called *query translation*, and the target is the continuous query language *SNEEql* (Brenninkmeijer, et al., 2008), which is expressive enough to deal with both streaming and stored sources. After the continuous query has been generated, the *Sensor NEtwork Engine (SNEE)* (Galpin et al., 2009) is used to evaluate the query over streaming and stored data sources. The result of the query processing is a set of tuples that the *data translation* process transforms into ontology instances.

2.3.4 CQELS

CQELS (Continuous Query Evaluation over Linked Streams) (Le-Phuoc et al., 2010; Le-Phuoc et al., 2011a; Le-Phuoc et al., 2011b) is currently the only *native and adaptive* query processor for unified query processing over both Linked Stream Data and Linked Data. Most of the current streaming RDF engines, e.g., C-SPARQL and $SPARQL_{Stream}$, use a "black box" approach, which delegates the processing to other engines, e.g., stream/event processing engines and/or SPARQL query processors. The continuous SPARQL queries supported by the streaming RDF engine are translated into the language(s) supported by the underlying engine(s). In contrast, CQELS uses a "white box" approach and implements the required query operators natively to avoid the overhead and limitation of closed system regime, and open up possibilities to add optimisation mechanisms to the kernel of the query processor. Therefore, CQELS has adopted many ideas from both the data stream and the database management systems, e.g., Eddies (Avnur and Hellerstein, 2000), dictionary encoding, intermediate results caching and hash indexing.

Inspired by Eddies, CQELS provides a flexible query execution framework with the query processor dynamically adapting to the changes in the input data. During query execution, it continuously reorders the operators according to some heuristics to achieve improved query execution in terms of delay and complexity. Much attention has paid to reduce disk I/O and memory consumption. Dictionary encoding is used to encode the strings in the RDF data. This strategy not only reduces disk I/O and memory consumption, but also speeds up operations on strings, such as pattern matching. Based on the observation that the Linked Data datasets rarely change, as soon as a query is registered, the output of the sub-queries over the Linked Data datasets is materialised and stored in a main memory cache that is available to the remaining query operations. The idea of materialising sub-query results is similar to that of the recycler

(Ivanova et al, 2009 and Ivanova et al, 2010), but with no sophisticated mechanisms to manage the cache yet. Hash indices are used to speed up lookup in the cache. In CQELS, when to create an index is decided as follows. Cache data is always indexed. For data coming from window operators, an index is maintained as long as it can be updated faster than the window's stream rate. If this threshold is reached, the index is dropped. The relational operators that depend on this index are replaced by equivalent ones that can work without indexes.

The authors have also introduced a declarative query language called the CQELS language by extending the SPARQL 1.1 (Harris and Seaborne, 2012) grammar under the EBNF notation. A query pattern, called `StreamGraphPattern`, is added into the `GraphPatternNotTriples` pattern to present window operators on RDF Stream. `StreamGraphPattern` is defined in such a way that it supports both time-based and triple-based windows and for time-based windows, a sliding parameter can be specified.

2.3.5 Streaming Knowledge Bases

The Streaming Knowledge Base (Walavalkar et al., 2008) tackles RDF data streams processing from a different angle than the aforementioned work, namely an approach for reasoning over streaming facts.

Reasoners built to handle static Semantic Web data, e.g., (Guo et al., 2004), perform poorly when used for reasoning over streaming facts. This is because those systems try to carry out the whole reasoning process, which involves computationally heavy inference methods like traversal of RDF graphs, at runtime. With streaming data, the incoming data rate is often much higher than the time taken by the reasoners for inference. To overcome this problem, Walavalkar et al. (2008) propose to split the reasoning process and try to pre-compute as mush as possible facts beforehand. Also, the authors try to leverage the work done by the DSMS and DBMS community by storing the pre-computed rules in database tables, using the engine to deal with streaming facts and using simple database queries to join the incoming stream with the rule tables to perform the inference. Therefore, the authors utilize the continuous query processor, TelegraphCQ (Chandrasekaran et al., 2003), to build a subsumption reasoner that can deal with streaming facts.

The operation of the streaming knowledge base is divided into two major functional components - *Ontology Pre-processor* and *Stream Processor*. The input to the ontology pre-processor is one or more OWL or RDFS ontology files. To process the ontology file, the InfModel class provided by the Jena toolkit (Carroll et al., 2004) is used to compute the five relationships: `rdfs:subClassOf`, `rdfs:subPropertyOf`, `rdfs:range`, `rdfs:domain` and `owl:inverseOf`, which are stored in tables in the TelegraphCQ database.

In addition to the basic subclassing, more complex inferences are supported in the query. For example, if a standing query asks for instances of class "man", and an incoming triple says that "John is the father of Mary", then "John" should satisfy this query. To achieve this, an intermediate stream handler is implemented to peek at the data stream and insert into it some inferred triples. For instance, if there is an incoming triple <*sub*, *pred*, *obj*> and from the ontology it is know that <*pred*, `rdfs:range`, *x*>, then the handler will insert a new triple <*obj*, `rdfs:type`, *x*>. Moreover, if *x* is the class of concern or a subclass of the class of concern, *obj* will be detected as an instance of concern, based on the relationships computed by the Ontology Pre-processor, even though the triple <*obj*, `rdfs:type`, *x*> never appeared in the incoming stream. This technique is also used to infer tuples based on the knowledge one can deduce from `rdfs:domain`, `rdfs:subPropertyOf` and `owl:inverseOf`. An interesting contribution of the Streaming Knowledge Base is the expressing of some inference rules in SQL queries.

2.3.6 Other Streaming RDF Techniques

Semantic Streams (Whitehouse et al., 2006) was among the first systems to propose semantic processing of streams. It uses Prolog-based logic rules to allow users to pose declarative queries over semantic interpretations of sensor data.

Semantic System S (Bouillet et al., 2007) proposes the use of the Web Ontology Language (OWL) to represent sensor data streams, as well as processing elements for composing applications from input data streams.

The Semantic Sensor Web project (Balazinska et al., 2007; Sheth et al., 2008) also focuses on interoperability between different sensor sources, as well as providing contextual information about the data. It does so by annotating sensor data with spatial, temporal, and thematic semantic metadata.

Proposals, such as the one from the *W3C Semantic Sensor Network Incubator Group*[28], aim at the integration of stream data with Linked Data sources by following the Linked Data principles for representing the data. In parallel, the concept of Linked Stream Data was introduced (Sequeda and Corcho 2009), in which URIs were suggested for identifying sensors and stream data.

Groppe et al. (2007) firstly propose a *streaming SPARQL engine*, which evaluates SPARQL queries on streams of RDF data. In this work, the authors define a set of Streaming SPARQL algebraic operators, including `Stream`, $\texttt{Match}_{\texttt{Pats}}$, $\texttt{Pat}_{(\texttt{p1, p2,p3})}$, `Join`, $\texttt{Sel}_{\texttt{expression}}$, `Optional`, $\texttt{Projection}_{\texttt{v}}$, `Distinct`, `Union`, and `Output`. In this work, static data are turned into data streams for better query performance, so it does not deal with real data streams (i.e., those produced by sensors and other mobile devices). Initially, work has only been done at the algebraic lever. An extension to the SPARQL query language to process data streams is added later in (Groppe, 2011).

Hoeksema (2011) presents a *distributed approach for Stream Reasoning*, i.e., processing C-SPARQL queries using the Yahoo S4 platform[29]. The author shows how the computation of the RDFS closures of an RDF stream and the components to perform C-SPARQL processing can be implemented using networks of S4's Processing Elements (PEs).

Event Processing SPARQL (EP-SPARQL) (Anicic et al., 2011b) is a language to describe event processing and stream reasoning and it can be translated to ETALIS (Anicic et al., 2011a), a Prolog-based complex event processing framework. First, RDF-based data elements are transformed into logic facts, and then EP-SPARQL queries are translated into Prolog rules.

2.3.7 Discussion

In general, the streaming RDF processing engines take a similar approach. First, a notion of time is added to the RDF data model, so that data streams can be represented as RDF triples according to some ontology. Then, the SPARQL query language is extended with new syntax and/or operators to enable continuous queries on RDF data streams. The extensions are typically inspired by the continuous query languages from the DSMSs. Finally, a query processor is provided to handle the continuous SPARQL queries. The main differences of the systems discussed in this section lay on the extension to the SPARQL language, which we will discuss in more details below, and the query processing engine(s) used. For this later issue, most systems rely on an existing DSMS to handle streaming data, possibly accompanied by an existing SPARQL engine to RDF data. CQUELS is the only native streaming RDF engine. Additionally, several systems spend efforts on the issue of ontology-based inference of new data from the streaming data.

The major work on extending SPARQL for streaming data processing includes StreamSPARQL (Bolles et al., 2008), C-SPARQL (Barbieri et al., 2010b), $\text{SPARQL}_{\text{Stream}}$ (Calbimonte et al., 2010) and CQUELS (Le-Phuoc et al., 2011a). They all add RDF data stream semantics to RDF/SPARQL, and they all support sliding windows. They only have small, but sometimes important, differences in the exactly supported language features. StreamSPARQL is the only extension allowing windows in graph patterns, but does not support aggregations. $\text{SPARQL}_{\text{Stream}}$ is the only extension that does not support element-based windows, but does support time-based windows in the past, and the window-to-stream operators originated from CQL. C-SPARQL is the only extension that supports stream and query registrations. C-SPARQL, $\text{SPARQL}_{\text{Stream}}$ and CQUELS are based on SPARQL 1.1, while StreamSPARQL is based on SPARQL 1.0. As a result of this, StreamSPARQL does not have aggregations.

In this deliverable, we have chosen to evaluate SRBench firstly on $\text{SPARQL}_{\text{Stream}}$, because compared with CQUELS, the language extension is cleaner and the architecture of the query processor is more flexible. The implementation of $\text{SPARQL}_{\text{Stream}}$ is module-based, which makes it easy to extend the system with a new

[28] See: http://www.w3.org/2005/Incubator/ssn/

[29] See: http://incubator.apache.org/s4/

module to use MonetDB/DataCell[30] as the continuous query processor. However both C-SPARQL and CQUELS are good alternatives, on which one can apply SRBench for performance comparison in the future.

2.4 RDF/SPARQL Benchmarks

With the growth and the availability of many systems supporting RDF/SPARQL, increasing efforts have been made in developing benchmarks for evaluating the performance of RDF stores. The representatives of the RDF benchmarks that have been widely used are the Lehigh University Benchmark (LUBM) (Guo et al., 2005), the Berlin SPARQL Benchmark (BSBM) (Bizer and Schultz 2009), and the SPARQL Performance Benchmark (SP^{2}Bench) (Schmidt et al., 2009).

2.4.1 LUBM

The Lehigh University Benchmark (LUBM) (Guo et al., 2005) is one of the first RDF benchmarks. It is built over a university domain in order to mainly evaluate the reasoning capability and inference mechanism of OWL (Web Ontology Language) Knowledge Base Systems. However, the generated dataset is quite homogenous without considering skewed data distributions as well as realistic correlations. Besides, as the testing queries of this benchmark are plain and lacking of important SPARQL features such as `FILTER` and `UNION`, it, therefore, cannot be used for evaluating the performance of the tested systems in supporting SPARQL features.

2.4.2 SP^{2}Bench

The SPARQL Performance Benchmark (SP^{2}Bench) (Schmidt et al., 2009) uses the Digital Bibliography & Library Project (DBLP)[31] as its domain and generates the synthetic dataset mimicking the original DBLP data. Observing the drawbacks of LUBM, this benchmark supports various features of SPARQL such as FILTER. However, as the matter of the SPARQL 1.1's availability at that moment, features like Aggregation, Subqueries, and Property paths, which were added in SPARQL 1.1, do not appear in the benchmark queries. In this work, the authors have tried to incorporate several realistic distributions such as the power-law distributions for the number of publications per author as well as the number of citations per paper. This may generate realistic degree distributions in RDF graph; however, as some important characteristics of "small-world" network graph such as the clustering coefficient are not considered, the graph complexity of generated data is quite limited. Besides, semantics correlations of realistic data are hardly addressed in SP^{2}Bench. In addition, due to its limited schema with only eight document classes, the SP^{2}Bench's generated data is lack of heterogeneity.

2.4.3 BSBM

The Berlin SPARQL Benchmark (BSBM) (Bizer and Schultz 2009) is probably the current most popular RDF/SPARQL benchmark. It is built around an e-commerce use case where products are offered by various vendors and get the reviews from various customers in different review sites. In this benchmark, the authors emulate the realistic search and navigation pattern of a customer by running the benchmark test driver over several query mixes (i.e., sequences of queries with different frequencies). By now, the benchmark has been used for evaluating a number of RDF stores[32] including Virtuoso[33], BigOwlim[34], Jena-TDB[35], 4store[36], and BigData[37]. However, the main drawback of BSBM is that with the homogenous relational-like schema and

[30] MonetDB/DataCell is the objective of the deliverable D1.7 of PlanetData (due in month 42).

[31] See: http://dblp.uni-trier.de/

[32] BSBM results: http://www4.wiwiss.fu-berlin.de/bizer/BerlinSPARQLBenchmark/results/V6/index.html

[33] See: http://virtuoso.openlinksw.com/

[34] See: http://www.ontotext.com/owlim/

[35] See: http://www.openjena.org/TDB/

[36] See: http://4store.org/

[37] See: http://www.systap.com/bigdata.htm

workload, it can be considered as a straight transformation from such relational benchmark as TPC-H[38]. It, thus, does not give insightful evaluations on the performance of RDF stores in comparing with a relational DBMS. Besides, since BSBM does not consider realistic data correlations as well as RDF graph complexity, no advanced features of RDF representation can be taken into account in running this benchmark.

2.4.4 Discussion

In a nutshell, the existing benchmarks are mostly relational-like, lack heterogeneity or are limited in representing realistic skewed data distributions and correlations. No new features of SPARQL 1.1, such as property path expression have been addressed in these benchmarks. Besides, although one advantage of RDF is the flexibility in sharing and integrating linked open knowledge bases, existing benchmarks solely work with one generated dataset without exploiting the knowledge from other linked open data such as DBpedia.

[38] Transaction Processing performance Council (TPC). See: http://www.tpc.org/

3 Data Sources

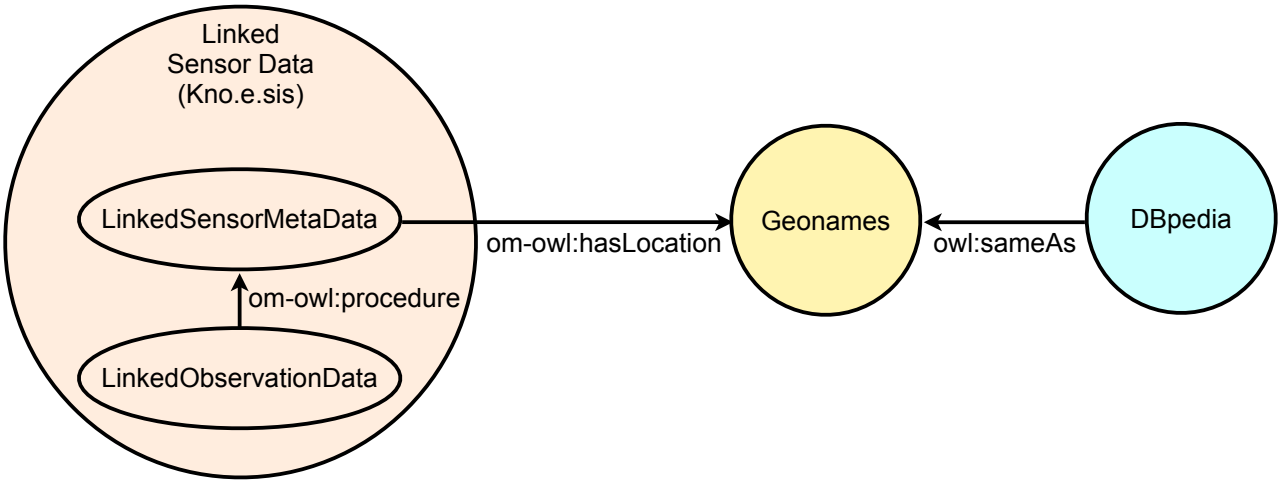

Figure 5: An overview of the datasets used in SRBench, and their relationships. (Note that the sizes of the circles do not reflect the actual sizes of the datasets.)

SRBench uses three real world datasets, i.e., the LinkedSensorData dataset[17], the RDF version of the GeoNames dataset[15] and the DBpedia dataset[16], as the basics of the data used by the benchmark. These three datasets were purposely chosen. The LinkedSensorData dataset publishes sensor metadata and sensor observation data according to the Linked Stream Data principle (Sequeda and Corcho, 2009), which was studied in the previous PlanetData deliverable D1.1 (Corcho et al., 2011). Moreover, LinkedSensorData is the largest sensor dataset in the Linked Open Data cloud[18] and the CKAN data portal[19]. The LinkedSensorData dataset links the sensor locations to nearby geographic places defined by the GeoNames dataset, so this naturally determines our choice of the GeoNames dataset. The choice for the DBpedia dataset is also a matter of course, since DBpedia is the largest and the most popularly used dataset in the Linked Open Data cloud. Moreover, the PlanetData partner FUB is one of the originators of DBpedia. An overview of the datasets is shown in Figure 5. In this section, we give some brief background information of these datasets.

3.1 The LinkedSensorData Dataset

Work on producing Linked Data from data emitted by sensors was initiated in 2009, pioneered by (Sequeda and Corcho, 2009) and (Le-Phuoc and Hauswirth, 2009). The *Linked Stream Data* principle is introduced by (Sequeda and Corcho, 2009) as the application of the Linked Data principles to sensor generated data.

The LinkedSensorData dataset is provided by Kno.e.sis. The LinkedSensorData are real-world dataset containing the US weather data collected by MesoWest[39], a project within the Department of Meteorology at the University of Utah that has been aggregating weather data since 2002. The data were turned into Linked Stream Data by the Semantic Sensor Web[40] and STT[41] projects at Kno.e.sis[42]. The LinkedSensorData is the first sensor dataset in the Linked Open Data cloud, and so far, the largest Linked Stream Data dataset in both LOD[18] and CKAN[19] containing ~1.7 billion triples. The LinkedSensorData dataset contains two sub-datasets, one for sensor metadata and another for sensor observation data.

[39] See: http://mesowest.utah.edu/index.html

[40] See: http://wiki.knoesis.org/index.php/SSW

[41] See: http://knoesis.org/research/semweb/projects/stt/

[42] See: http://knoesis.wright.edu

The *LinkedSensorMetadata*[43] is an RDF dataset containing expressive descriptions of ~20,000 weather stations in the United States. The data originated at the MesoWest[39] project. On average, there are about five sensors per weather station, so there are in total ~100,000 sensors in the dataset, described according to the *sensor-observation ontology*[44]. The sensors measure phenomena such as temperature, visibility, precipitation, pressure, wind speed, humidity, etc. In addition to location attributes such as latitude, longitude, and elevation, there are also links to locations in *GeoNames* that are near each weather station.

The *LinkedObservationData* is an RDF dataset containing expressive descriptions of hurricane and blizzard observations in the United States. The data originated at the MesoWest[39] project. The observations collected include measurements of phenomena such as temperature, visibility, precipitation, pressure, wind speed, humidity, etc. The dataset includes observations within the entire United States during the time periods that several major storms were active – including Hurricane Katrina, Ike, Bill, Bertha, Wilma, Charley, Gustav, and a major blizzard in Nevada in 2003. These observations are generated by weather stations described in the LinkedSensorMetadata dataset introduced above. Currently, this dataset contains almost two billion RDF triples, which together describe more than 159 million observations. For SRBench, we have obtained all linked sensor observation datasets from the original Kno.e.sis site for LinkedSensorData17.

Table 1 shows the statistics of the LinkedObservationData datasets as presented on the original website, to which we have added the sizes of the datasets after they have been unpacked.

Table 1: Statistics of the Sensor Observation Datasets Used by SRBench

Name	Storm Type	Date	#Triples	#Observations	Data size
ALL			1,730,284,735	159,460,500	~111 GB
Bill	Hurricane	Aug. 17 – 22, 2009	231,021,108	231,021,108	~15 GB
Ike	Hurricane	Sep. 1 – 13, 2008	374,094,660	34,430,964	~34 GB
Gustav	Hurricane	Aug. 25 – 31, 2008	258,378,511	23,792,818	~17 GB
Bertha	Hurricane	Jul. 6 – 17, 2008	278,235,734	25,762,568	~13 GB
Wilma	Hurricane	October 17 – 23, 2005	171,854,686	15,797,852	~10 GB
Katrina	Hurricane	August 23 – 30, 2005	203,386,049	18,832,041	~12 GB
Charley	Hurricane	August 9 – 15, 2004	101,956,760	9,333,676	~7 GB
	Blizzard	April 1 – 6, 2003	111,357,227	10,237,791	~2 GB

3.2 The GeoNames RDF Dataset

GeoNames[45] is a free geographical database covers all countries and contains over eight million place names that are available for download. Originally, the GeoNames dataset was only available in tab-delimited text in utf8 encoding. With the introduction of the GeoNames Ontology in 2006 by Bernard Vatant, the GeoNames dataset can be expressed in RDF for the Semantic Web. Currently, more than 6.2 million geo-names toponyms now have a unique URL with a corresponding RDF web service.

For SRBench, we use version 3.01 of the GeoNames ontology[46]. We have obtained the dump of the complete GeoNames RDF dataset, which contains about 8 million geographic features with about 146 million RDF

[43] In the original website17, both the complete dataset and the sub-dataset containing the sensor metadata are referred to as "linked sensor data". To avoid confusing, in this deliverable, we refer to the sub-dataset containing the sensor metadata as "LinkedSensorMetadata".

[44] http://knoesis.wright.edu/resources/library-resources/files/ontologies/sensor-ob/sensor-observation.owl

[45] See: http://www.geonames.org/

[46] See: http://www.geonames.org/ontology/ontology_v3.01.rdf

triples. The dump has one RDF document per toponym. The complete dataset takes about 10 Gigabytes on disk.

3.3 The DBpedia Dataset

The DBpedia dataset is the largest and most popularly used dataset in the Linked Open Data cloud. It is developed by, among others, the PlanetData partner FUB. DBpedia extracts various kinds of structured information from Wikipedia editions in 97 languages and combines this information into a huge, cross-domain knowledge base. Subsequently, this information is made available on the (Semantic) Web as RDF data. This way, DBpedia allows users to ask sophisticated (SPARQL) queries against Wikipedia information and to link other data sets on the Web to Wikipedia data. The DBpedia data set uses a large cross-domain ontology[47], which has been manually created based on the most commonly used infoboxes of Wikipedia. The ontology currently covers over 320 classes which form a subsumption hierarchy and are described by 1,650 different properties.

As of Version 3.7 (July 2011), the DBpedia knowledge base describes more than 3.64 million "things" with over half a billion "facts". Among all "things", 1.83 million of them are classified in a consistent Ontology, including 416,000 persons, 526,000 places (including 360,000 populated places), 106,000 music albums, 60,000 films, 17,500 video games, 169,000 organizations (including 40,000 companies and 38,000 educational institutions), 183,000 species and 5,400 diseases. The DBpedia data set features labels and abstracts for these 3.64 million things in up to 97 different languages; 2,724,000 links to images and 6,300,000 links to external web pages; 6,200,000 external links into other RDF datasets, 740,000 Wikipedia categories, and 2,900,000 YAGO categories. The dataset consists of 1 billion pieces of information (RDF triples) out of which 385 million were extracted from the English edition of Wikipedia and roughly 665 million were extracted from other language editions and links to external datasets.

Each thing in the DBpedia data set is identified by a URI reference of the form *http://dbpedia.org/resource/Name*, where *Name* is taken from the URL of the source Wikipedia article, which has the form *http://en.wikipedia.org/wiki/Name*. Thus, each resource is tied directly to an English-language Wikipedia article. Every DBpedia resource is described by a label, a short and long English abstract, a link to the corresponding Wikipedia page, and a link to an image depicting the thing (if available). If a thing exists in multiple language versions of Wikipedia, then short and long abstracts within these languages and links to the different language Wikipedia pages are added to the description.

For SRBench, we have only obtained the datasets from the English language collection, which consists of 44 RDF files in N-triple format with about 181 million triples. The total data size is about 27 Gigabytes. The DBpedia dataset is directly linked to the GeoNames dataset through the `owl:sameAs` property. DBpedia has in total 85,000 links to the GeoNames dataset. For instance, the DBpedia resource about Cambridge (http://dbpedia.org/resource/Cambridge) describes the same thing as the GeoNames feature 2653941 (http://sws.geonames.org/2653941/). The DBpedia dataset is not directly linked to the LinkedSensorData dataset.

[47] See: http://wiki.dbpedia.org/Ontology?v=181z

4 SRBench Dataset Specification

4.1 Namespaces

@prefix	category:	<http://dbpedia.org/resource/Category:> .
@prefix	cc:	<http://creativecommons.org/ns#> .
@prefix	dbp:	<http://dbpedia.org/ontology/> .
@prefix	dbpprop:	<http://dbpedia.org/property/> .
@prefix	dcterms:	<http://purl.org/dc/terms/> .
@prefix	foaf:	<http://xmlns.com/foaf/0.1/> .
@prefix	gn:	<http://www.geonames.org/ontology#> .
@prefix	om-owl:	<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#> .
@prefix	owl:	<http://www.w3.org/2002/07/owl#> .
@prefix	owl-time:	<http://www.w3.org/2006/time#> .
@prefix	rdf:	<http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix	rdfs:	<http://www.w3.org/2000/01/rdf-schema#> .
@prefix	sens-obs:	<http://knoesis.wright.edu/ssw/> .
@prefix	skos:	<http://www.w3.org/2004/02/skos/core#> .
@prefix	srb:	<http://www.cwi.nl/srbench/> .
@prefix	time:	<http://www.w3.org/2006/time> .
@prefix	weather:	<http://knoesis.wright.edu/ssw/ont/weather.owl#> .
@prefix	wgs84_pos:	<http://www.w3.org/2003/01/geo/wgs84_pos> .
@prefix	xsd:	<http://www.w3.org/2001/XMLSchema#> .
@prefix	yago:	<http://dbpedia.org/class/yago/> .

4.2 Classes and Properties

In this section, we describe the classes and their properties as defined by the sensor-observation ontology[44], and the GeoNames ontology[46]. An overview of the ontology classes and how there are linked to each other are shown in Figure 6. The DBpedia ontology[47] is highly complex but well documented, so we do not repeat its class definitions here.

Class `System`

The ontology class `System` describes a weather sensor station. Each weather sensor station has several properties, e.g., an ID, the location of the station, a geographical location to which the station is located nearby, the weather properties observed by this station and the original MesoWest URL of the station.

- `rdf:type` (resource: `om-owl:System`)
- `om-owl:ID` (literal: String)
- `om-owl:hasLocatedNearRel` (resource: `sens-obs:LocatedNearRel`, nearby location)
- `om-owl:hasSourceURI` (resource: MesoWest URI)
- `om-owl:parameter` (resource: `om-owl:PropertyType`, properties observed by this sensor which can be any number of the weather properties defined by the weather ontology):
  - `weather:_AirTemperature`
  - `weather:_DewPoint`
  - `weather:_PeakWindDirection`

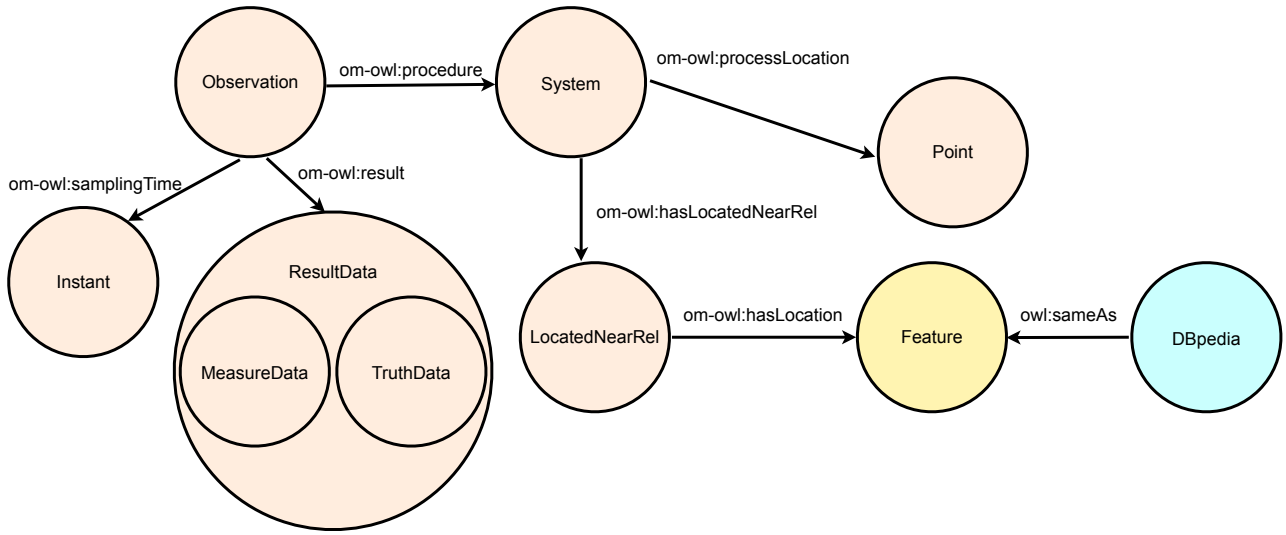

Figure 6: An overview of all ontology classes and their relationships.

- `weather:_PeakWindSpeed`
- `weather:_PrecipitationAccumulated`
- `weather:_PrecipitationSmoothed`
- `weather:_RelativeHumidity`
- `weather:_SnowDepth`
- `weather:_SnowInterval`
- `weather:_SnowSmoothed`
- `weather:_SoilMoisture`
- `weather:_SoilTemperature`
- `weather:_Visibility`
- `weather:_WindDirection`
- `weather:_WindGust`
- `weather:_WindSpeed`

- `om-owl:processLocation` (resource: `wgs84_pos:Point`)

Example RDF instance:

```
sens-obs:System_A24
 a om-owl:System ;
 om-owl:ID "A24" ;
 om-owl:hasLocatedNearRel sens-obs:LocatedNearRelA24 ;
 om-owl:hasSourceURI
  <http://mesowest.utah.edu/cgi-bin/droman/meso_base.cgi?stn=A24> ;
 om-owl:parameter weather:_WindDirection ,
          weather:_RelativeHumidity ,
          weather:_DewPoint ,
          weather:_AirTemperature ,
          weather:_WindSpeed ,
          weather:_WindGust ;
 om-owl:processLocation sens-obs:point_A24 .
```

Class LocatedNearRel

The ontology class `LocatedNearRel` describes a geographic location to which a weather sensor station is located nearby. In this class, properties, such as the distance between the nearby location and the sensor station, are defined.

- `rdf:type` (resource: `om-owl:LocatedNearRel`)
- `om-owl:hasLocation` (resource: GeoNames URI, location of the nearby place defined by GeoNames)
- `om-owl:distance` (literal: `xsd:float`)
- `om-owl:uom` (resource: `om-owl:UnitOfMeasurement`, whose actual value is one of the units of measurement defined in the sensor observation ontology and the weather ontology):
  - `om-owl:degrees`
  - `om-owl:percent`
  - `weather:centimeters`
  - `weather:fahrenheit`
  - `weather:inches`
  - `weather:meters`
  - `weather:miles`
  - `weather:milesPerHour`

Example RDF instance:

```
sens-obs:LocatedNearRelA24
 a om-owl:LocatedNearRel ;
 om-owl:distance "2.1431"^^xsd:float ;
 om-owl:hasLocation <http://sws.geonames.org/5512628/> ;
 om-owl:uom weather:miles .
```

Class `Point`

The ontology class `Point` describes a geographic point location, determined by its latitude, longitude and altitude.

- `rdf:type` (resource: `wgs84_pos:Point`)
- `wgs84_pos:lat` (literal: `xsd:float`, latitude in degrees)
- `wgs84_pos:long` (literal: `xsd:float`, longitude in degrees)
- `wgs84_pos:alt` (literal: `xsd:float`, elevation in meters)

Example RDF instance:

```
sens-obs:point_A24
 a wgs84_pos:Point ;
 wgs84_pos:alt "4951"^^xsd:float ;
 wgs84_pos:lat "36.8433"^^xsd:float ;
 wgs84_pos:long "-116.47"^^xsd:float .
```

Class Observation

The ontology class `Observation` describes a observation made by a weather sensor station. Each observation has several properties, such as the ID of the weather station that has made the observation, the type of the observed weather property, the value of the observation and the time when the observation was made.

- `rdf:type` (resource: `om-owl:Observation`, whose actual value is one of the subclasses of `om-owl:Observation` defined by the weather ontology to denote weather observations):
  - `weather:WindSpeedObservation`
  - `weather:PrecipitationObservation`
  - `weather:RainfallObservation`
  - `weather:SnowfallObservation`
  - `weather:PressureObservation`
  - `weather:RadiationObservation`
  - `weather:TemperatureObservation`
  - `weather:FreezingTemperatureObservation`
- `om-owl:featureOfInterest` (resource: `om-owl:Feature`)
- `om-owl:observationLocation` (resource: `wgs84_pos:Location`)
- `om-owl:memberOf` (resource: `om-owl:ObservationCollection`)
- `om-owl:observedProperty` (resource: `om-owl:PropertyType`, which actual value is one of the weather properties defined by the weather ontology, as listed under the `om-owl:parameter` property of the class `sens-obs:System` above)
- `om-owl:procedure` (resource: `om-owl:System`, refers to the sensor generated this observation)
- `om-owl:result` (resource: `om-owl:ResultData`)
- `om-owl:resultTime` (resource: `time:Time`)
- `om-owl:samplingTime` (resource: `om-owl:Instant`)

Example RDF instance:

```
sens-obs:Observation_AirTemperature_A24_2004_8_10_21_00_00
 a weather:TemperatureObservation ;
 om-owl:observedProperty weather:_AirTemperature ;
 om-owl:procedure sens-obs:System_A24 ;
 om-owl:result sens-obs:MeasureData_AirTemperature_A24_2004_8_10_21_00_00 ;
 om-owl:samplingTime sens-obs:Instant_2004_8_10_21_00_00 .
```

Class MeasureData

The ontology class `MeasureData` describes the numerical value of an observation.

- `rdf:type` (resource: `om-owl:MeasureData`, subclass of `om-owl:ResultData`)
- `om-owl:uom` (resource: `om-owl:UnitOfMeasurement`, whose actual value is one of the units of measurement defined in the sensor observation ontology and the weather ontology, as listed under the `om-owl:uom` property of the class `sens-obs:LocatedNearRel` above)
- `om-owl:floatValue` (literal: `xsd:float`)

Example RDF instance:

```
sens-obs:MeasureData_AirTemperature_A24_2004_8_10_21_00_00
 a om-owl:MeasureData ;
 om-owl:floatValue "91.0"^^xsd:float ;
 om-owl:uom weather:fahrenheit .
```

Class TruthData

The ontology class `TruthData` describes the truth-value of an observation.

- `rdf:type` (resource: `om-owl:TruthData`, subclass of `om-owl:ResultData`)
- `om-owl:booleanValue` (literal: `xsd:boolean`)

Example RDF instance:

```
sens-obs:TruthData_SnowSmoothed_ABRM8_2004_8_9_11_00_00
 a om-owl:TruthData ;
 om-owl:booleanValue "true^^http://www.w3.org/2001/XMLSchema#boolean" .
```

Class Instant

The ontology class `Instant` describes a date/time object.

- `rdf:type` (resource: `owl-time:Instant`)
- `owl-time:inXSDateTime` (literal: `xsd:dateTime`)

Example RDF instance:

```
sens-obs: Instant_2004_8_10_21_00_00
 a owl-time:Instant ;
 owl-time:inXSDDateTime
  "2004-08-10T21:00:00-07:00^^http://www.w3.org/2001/XMLSchema#dateTime" .
```

Class `Feature`

The ontology class `Feature` describes a geographical location. Each observation has several properties, such as different types of names of the location, the latitude and longitude of the location, the country to which this location belong and other locations that are close to this location.

- `rdf:type` (resource: `gn:Feature`, a geographical object uniquely defined by its GeoNames ID.)
- `gn:countryCode` (literal: String, a two letters country code in the ISO 3166 list)
- `gn:name` (literal: String, the main international name of a feature. The value has no `xml:lang` tag.)
- `gn:officialName` (literal: String, a name in an official local language)
- `gn:population` (literal: `xsd:integer`)
- `gn:postalCode` (literal: String)
- `gn:shortName` (literal: String)
- `gn:childrenFeatures` (resource: `gn:RDFData`, links to an RDF document containing the description of children features)
- `gn:featureClass` (resource: `gn:Class`, the main category of the feature, as defined in GeoNames taxonomy)
- `gn:featureCode` (resource: `gn:Code`, type of the feature, as defined in GeoNames taxonomy)
- `gn:locationMap` (resource: `gn:Map`, a GeoNames map centered on the feature)
- `gn:nearby` (resource: `gn:Feature`, a feature close to the reference feature)
- `gn:nearbyFeatures` (resource: `gn:RDFData`, links to an RDF document containing the descriptions of nearby features)
- `gn:neighbour` (resource: `gn:Feature`, a feature sharing a common boarder with the reference feature)

- `gn:neighbouringFeatures` (resource: `gn:RDFData`, links to an RDF document containing the descriptions of neighbouring features. Applies when the feature has definite boarders.)
- `gn:parentADM{1,2,3,4}` (resource: `gn:featureCode`, level 1, 2, 3 or 4 administrative parent, as defined in the GeoNames ontology)
- `gn:parentCountry` (resource: `gn:featureCode`)
- `gn:parentFeature` (resource: `gn:Feature`)
- `gn:wikipediaArticle` (resource: `gn:WikiPediaArticle`, URL of a Wikipedia article of which subject is the resource)
- `wgs84_pos:lat` (literal: `xsd:float`, latitude in degrees)
- `wgs84_pos:long` (literal: `xsd:float`, longitude in degrees)

Example RDF instance:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<rdf:RDF xmlns:cc="http://creativecommons.org/ns#"
     xmlns:dcterms="http://purl.org/dc/terms/"
     xmlns:foaf="http://xmlns.com/foaf/0.1/"
     xmlns:gn="http://www.geonames.org/ontology#"
     xmlns:owl="http://www.w3.org/2002/07/owl#"
     xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
     xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
     xmlns:wgs84_pos="http://www.w3.org/2003/01/geo/wgs84_pos#">
 <gn:Feature rdf:about="http://sws.geonames.org/4045829/">
  <rdfs:isDefinedBy>
   http://sws.geonames.org/4045829/about.rdf
  </rdfs:isDefinedBy>
  <gn:name>Omik</gn:name>
  <gn:featureClass rdf:resource="http://www.geonames.org/ontology#S"/>
  <gn:countryCode>US</gn:countryCode>
  <wgs84_pos:lat>52.41667</wgs84_pos:lat>
  <wgs84_pos:long>173.58333</wgs84_pos:long>
  <gn:parentFeature rdf:resource="http://sws.geonames.org/5879164/"/>
  <gn:parentCountry rdf:resource="http://sws.geonames.org/6252001/"/>
  <gn:parentADM1 rdf:resource="http://sws.geonames.org/5879092/"/>
  <gn:parentADM2 rdf:resource="http://sws.geonames.org/5879164/"/>
  <gn:nearbyFeatures
   rdf:resource="http://sws.geonames.org/4045829/nearby.rdf"/>
  <gn:locationMap rdf:resource="http://www.geonames.org/4045829/omik.html"/>
 </gn:Feature>
</rdf:RDF>
```

4.3 Data dictionaries

Only one dictionary is used in the benchmark to generate sensor locations.

- Dictionary 1: locations (latitude, longitude, altitude) of all sensors in the LinkedSensorMetadata dataset, file: sensor_locations.nt

5 SRBench Query Definitions

In this section, we give the definitions of the SRBench benchmark queries. The benchmark contains a concise, yet comprehensive set of 17 queries that covers the major aspects of streaming RDF/SPARQL processing, and each query is intended to challenge a particular aspect of the query processor. Since there exists no standard query language for streaming RDF data, the query requirements for a streaming RDF benchmark should be language-agnostic, yet have a clear semantics. As a result of this, all SRBench queries are defined in terms of the knowledge one wants to obtain or derive from the ontologies, rather than a specific streaming RDF query language.

The addressed RDF/SPARQL features range from simple pattern matching queries to queries with complex reasoning tasks. The main advantages of applying Semantic Web technologies on streaming data include providing better search facilities by adding semantics to the data, reasoning through ontologies, and integration with other data sets. The ability of a streaming RDF engine to process these distinctive features is accessed by the benchmark with queries that apply reasoning not only over the streaming sensor data, but also over the metadata and even other data sets in the Linked Open Data cloud, currently including the GeoNames RDF dataset[15] and the DBpedia dataset[16]. Thus, the SRBench benchmark queries are divided into three classes:

- Q1 – Q7 only query the dynamic streaming data;
- Q8 – Q11 query both the dynamic streaming data and the static sensor metadata;
- Q12 – Q17 query not only the Linked Sensor Data dataset but also the GeoNames and DBpedia datasets.

5.1 Basic Pattern Matching

Q1. Get all rainfall observed in the last hour.

Use case motivation: a basic but important query. It tests the engine's ability to handle the most basic feature of RDF/SPARQL to gain knowledge about the mostly spoken topic when talking about the weather.

Inputs: None

5.2 Optional Pattern Matching

Q2. Get all precipitation observed in the last hour.

Use case motivation: we go a step further and ask for all types of precipitation. Since the triple patterns for different kinds of precipitations maybe different, optional patterns are needed to capture the possible differences. Additionally, this query requires reasoning over all instances of the class `PrecipitationObservation` and its subclasses.

Inputs: None

5.3 `ASK` Query Form

Q3. Detect if a station is observing a hurricane.

Tips: a hurricane has a sustained wind (for more than 3 hours) of at least 33 metres per second or 74 miles per hour (119 km/h).

Use case motivation: we want to know if there are any extreme weather conditions among the observations. This query tests the engine's ability to filter out the minimal amount of the streaming data to quickly compute the answer.

Inputs: None

5.4 Overlapping Sliding Window

Q4. Get the average wind speed at the stations where the air temperature is >32 degrees in the last hour, every 10 minutes.

Use case motivation: combine values observed for multiple weather properties. This query tests the engine's ability to deal with data that need to be temporarily stored.

Inputs: None

5.5 `CONSTRUCT` Derived Knowledge

Q5. Detect if a station is observing a blizzard.

Tips: a blizzard is a severe snow storm characterised by low temperatures, strong winds and heavy snow lasting for at least three hours.

Use case motivation: detect extreme weather conditions by combining multiple observed weather properties. This query tests the engine's ability to produce new knowledge derived by combining existing data.

Inputs: None

5.6 Union

Q6. Get the stations that have observed extremely low visibility in the last hour.

Tips: next to direct measurements of low visibility (<10 centimeters), heavy snowfall and rainfall (> 30 centimeters) also cause low visibility.

Use case motivation: this is a more complex example of detecting extreme weather conditions, which requires not only gaining knowledge explicitly contained in the data, but also deriving implicit knowledge from data sources.

Inputs: None

5.7 Window-to-Stream operation

Q7. Detect stations that are recently broken.

Tips: if a station suddenly stops producing (observation) data, it might be broken.

Use case motivation: knowing the stability of the stations is an important issue, which can be deduced from absent data. This query tests the engine's ability to cope with the dynamic properties that are specific for streaming data.

Inputs: None

5.8 Aggregates

Q8. Get the daily minimal and maximal air temperature observed by the sensor at a given location.

Use case motivation: temperature is the most common weather condition queried. This query tests the engine's ability to aggregates data that are grouped by their geo-spatial properties.

Inputs:

Input	Description
%Latitude%	Latitude of the sensor
%Longitude%	Longitude of the sensor
%Altitude%	Altitude of the sensor

5.9 Expression in `SELECT` Clause

Q9. Get the daily average wind force and direction observed by the sensor at a given location.

Use case motivation: wind is the other most commonly asked weather condition. The Beaufort Wind Force Scale[48] is a better way to express the wind force than the wind speed, which requires some post processing of the qualified RDF triples. This query tests the engine's ability to post process the qualified triple patterns.

Inputs:

Input	Description
%Latitude%	Latitude of the sensor
%Longitude%	Longitude of the sensor
%Altitude%	Altitude of the sensor

5.10 Join

Q10. Get the locations where a heavy snowfall has been observed in the last day.

Use case motivation: we want to find places that are suitable for a ski holiday. This query tests the engine's ability to join the dynamic sensor streaming data with the static sensor metadata.

Inputs: None

5.11 Subquery

Q11. Detecting if a station is producing significantly different observation values than its neighbouring stations.

Tips: if two sensor stations are located close (i.e., `hasLocatedNearRel`) to the same location, these two sensors are neighbours of each other.

Use case motivation: we want to detect the malfunctioning sensors. This query tests the engine's ability to compute complex subquery.

Inputs: None

5.12 Property Path Expressions

This group of queries tests the engine's ability to derive knowledge from multiple interlinked datasets using Property Path expressions. In particular, the queries require computing paths with arbitrary lengths for the `parentFeature` relationship, and computing alternatives for the name of the resulting places.

Q12. Get the hourly average air temperature and humidity of large cities.

Tips: use the GeoNames dataset to find large cities, i.e., population > 15000, and use the `hasLocatedNearRel` property in the sensor ontology[44] to find sensors located in or near to these cities.

Use case motivation: we want to find out if the temperature is higher during the rush hours in large cities, which says something about the air pollution.

Inputs: None

Q13. Get the shores in Florida, US where a strong wind, i.e., the wind force is between 6 and 9, has been observed in the last hour.

Tips: first reason over the `parentADM{1,2,3,4}` and `parentFeature` properties of the GeoNames ontology to find the shores in Florida, US; and then use the `hasLocatedNearRel` property in the sensor ontology[44] to find sensors located near to these shores.

Use case motivation: we want to find shores in Florida, US where we can go windsurfing now.

Inputs: None

[48] See: http://en.wikipedia.org/wiki/Beaufort_scale

Q14. Get the airport(s) located in the same city as the sensor that has observed extremely low visibility in the last hour.

Tips: use the GeoNames dataset and the `hasLocatedNearRel` property in the sensor ontology[44] to find airport(s) and sensors located in the same city.

Use case motivation: we want to trigger an alarm if dangerous weather condition has been observed.

Inputs: None

5.13 Ontology-based Reasoning

This group of queries all tests the engine's ability to apply reasoning over the ontologies of the interlinked datasets. In particular, reasoning over `rdfs:subClassOf` is required to find all hurricanes; and reasoning over `owl:sameAs` is required to the same associate geographic features described in both the GeoNames RDF dataset and the DBpedia dataset.

Q15. Get the locations where the wind speed in the last hour is higher than a known hurricane.

Tips: use the DBpedia dataset to get the information of hurricanes in the past.

Use case motivation: by comparing it with historical values, we can detect extreme weather conditions.

Inputs: None

Q16. Get the heritage sites that are threatened by a hurricane.

Tips: use the DBpedia dataset to get the geographical information of the monuments; use the GeoNames dataset and the `hasLocatedNearRel` property in the sensor ontology to find sensors located close to the monuments.

Use case motivation: we want to trigger an alarm if dangerous weather condition has been observed.

Inputs: None

Q17. Estimate the damage where a hurricane has been observed.

Tips: use the DBpedia dataset to find the damage caused by earlier hurricanes in the same area.

Use case motivation: the first we want to know after a natural disaster is the damage it brings.

Inputs: None

5.14 Evaluation

When designing the queries, we have paid much attention to the coverage of the queries in different aspects. From the point of view of the users, the queries vary from simple queries with few semantics, to complex queries with richer semantics. The queries cover a broad scope of the knowledge that the users of a weather observation system would want to obtain given the ontologies.

From the point of view of RDF/SPARQL engines, the queries address a broad scope of features that are specific to RDF/SPARQL, which range from simple pattern matching queries to queries with complex reasoning tasks.

To the best of our knowledge, the Social Intelligence Benchmark (SIB)[49] is the only existing RDF/SPARQL benchmark that has adopted the SPARQL 1.1 feature "Property Path" expressions. However, the SRBench makes more extensive use of property path expressions for reasoning.

We have purposely chosen to not include the sorting feature in the benchmark query set, since it is an insignificant feature for streaming data. Because, on the one hand, streaming data are already sorted by nature by their timestamps; while on the other hand, requiring results to be sorted on another attribute will only produce partially sorted data (i.e., with *one* window).

[49] See: http://www.w3.org/wiki/Social_Network_Intelligence_BenchMark

We believe that the SRBench benchmark clearly shows the added values of the Semantic Web technologies on gaining more knowledge from streaming data. The benchmark provides a general framework to assess the ability of streaming RDF engines to support such applications.

6 SRBench Queries Implementation Using SPARQL$_{Stream}$

Streaming RDF processing is an evolving topic and the proposed systems are in their earlier stages of development. At this moment, one of the most important issues is to assess the functionality of the proposed systems. Do they provide a sufficient set of functions that are needed by the streaming applications? Do they miss any crucial functionalities? Do they provide any additional functionalities that can be beneficial for streaming applications, which thus distinguish themselves from other systems in the same area.

In the section, we report our experience on applying SRBench on the SPARQL streaming RDF extension and engine SPARQL$_{Stream}$, developed by the PlanetData partner UPM, for a functional evaluation. To answer the question "Is the SPARQL$_{Stream}$ streaming RDF processing engine functionally complete?", we have implemented the complete set of benchmark queries using the SPARQL$_{Stream}$ language extension. We choose SPARQL$_{Stream}$ as the first system to evaluate, because SPARQL$_{Stream}$ is the only SPARQL streaming language extension based on both SPARQL 1.1 and the CQL language. Additionally, the implementation of SPARQL$_{Stream}$ has a module structure, which can be easily extended to cooperate with other streaming data processors, e.g., MonetDB/DataCell[50].

The SPARQL$_{Stream}$ language extension is formally defined in (Calbimonte et al., 2010). To make this document self-containing, we have included the formal definition of the syntax and semantic of SPARQL$_{Stream}$ in Annex A. The queries have been executed using the SPARQL$_{Stream}$ engine, downloaded from http://code.google.com/p/semanticstreams/source/checkout.

6.1 Q1 – Get all rainfall observed in the last hour.

```
PREFIX om-owl: <http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#>
PREFIX weather: <http://knoesis.wright.edu/ssw/ont/weather.owl#>

SELECT DISTINCT ?sensor ?value ?uom
FROM NAMED STREAM <http://www.cwi.nl/SRBench/observations> [NOW - 1 HOURS]
WHERE {
  ?observation om-owl:procedure ?sensor ;
               a weather:RainfallObservation ;
               om-owl:result ?result .
  ?result om-owl:floatValue ?value ;
          om-owl:uom ?uom .
}
```

Query Properties:

- Use the `DISTINCT` solution modifier

6.2 Q2 – Get all precipitation observed in the last hour.

```
PREFIX om-owl: <http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX weather: <http://knoesis.wright.edu/ssw/ont/weather.owl#>

SELECT DISTINCT ?sensor ?value ?uom
FROM NAMED STREAM <http://www.cwi.nl/SRBench/observations> [NOW - 1 HOURS]
WHERE {
  ?observation om-owl:procedure ?sensor ;
               rdf:type/rdfs:subClassOf* weather:PrecipitationObservation ;
               om-owl:result ?result .
  ?result ?p1 ?value .
  FILTER( REGEX(STR(?p1), "value", "i") )
  OPTIONAL {
    ?result ?p2 ?uom .
```

[50] MonetDB/DataCell is a streaming data processing engine being developed by the PlanetData partner CWI. It will be part of the main objective of the following deliverable D1.7 of PlanetData.

```
    FILTER( REGEX(STR(?p2), "uom", "i") )
  }
}
```

Query Properties:

- Use the `DISTINCT` solution modifier
- Use `FILTER` constraints
- Use the `REGEX` and `STR` functions on strings
- Use an `OPTIONAL` graph pattern
- Use Property Path expression with an arbitrary length path, to reason over all instances of the class `weather:PrecipitationObservation` and its subclasses. Thus, an engine that supports subclass reasoning will not only return observations which is of the type `PrecipitationObservation`, but also observations which are of a subtype of `PrecipitationObservation`, e.g., `RainfallObservation` and `SnowfallObservation`.

6.3 Q3 – Detect if a station is observing a hurricane.

```
PREFIX om-owl: <http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#>
PREFIX weather: <http://knoesis.wright.edu/ssw/ont/weather.owl#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>

ASK
FROM NAMED STREAM <http://www.cwi.nl/SRBench/observations>
           [NOW - 3 HOURS SLIDE 10 MINUTES]
WHERE {
  ?observation om-owl:procedure ?sensor ;
               om-owl:observedProperty weather:WindSpeed ;
               om-owl:result [ om-owl:floatValue ?value ] .
}
GROUP BY ?sensor
HAVING ( AVG(?value) >= "74"^^xsd:float ) #milesPerHour
```

Query Properties:

- Use the `ASK` query form
- Use `GROUP BY`, `HAVING` and aggregation function
- Requires reasoning over instances of the property `weather:WindSpeed` and all its Subproperties. Thus, an engine that supports subproperty reasoning will not only return observations whose `observedProperty` is `WindSpeed`, but also observations whose `observedProperty` are a subproperty of `WindSpeed`, e.g., `PeakWindSpeed` and `WindGust`.
- Requires computing overlapping windows

6.4 Q4 – Get the average wind speed at the stations where the air temperature is >32 degrees in the last hour, every 10 minutes.

```
PREFIX om-owl: <http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#>
PREFIX weather: <http://knoesis.wright.edu/ssw/ont/weather.owl#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>

SELECT ?sensor ( AVG(?windSpeed) AS ?averageWindSpeed )
       ( AVG(?temperature) AS?averageTemperature )
FROM NAMED STREAM <http://www.cwi.nl/SRBench/observations>
           [NOW - 1 HOURS SLIDE 10 MINUTES]
WHERE {
  ?temperatureObservation om-owl:procedure ?sensor ;
```

```
                          a weather:TemperatureObservation ;
                          om-owl:result ?temperatureResult .
  ?temperatureResult om-owl:floatValue ?temperature ;
                     om-owl:uom ?uom .
  FILTER(?temperature > "32"^^xsd:float && REGEX(STR(?uom), "fahrenheit", "i"))
  ?windSpeedObservation om-owl:procedure ?sensor ;
                        a weather:WindSpeedObservation ;
                        om-owl:result [ om-owl:floatValue ?windSpeed ] .
}
GROUP BY ?sensor
```

Query Properties:

- Use expressions in the `SELECT` clause
- Use `FILTER` constraints
- Use the `REGEX` and `STR` functions on strings
- Use `GROUP BY` and aggregation functions
- Requires a join between two types of observations on the sensor ID.
- Requires computing overlapping windows

6.5 Q5 – Detect if a station is observing a blizzard.

```
PREFIX om-owl: <http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#>
PREFIX weather: <http://knoesis.wright.edu/ssw/ont/weather.owl#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>

CONSTRUCT { ?sensor om-owl:generatedObservation [ a weather:Blizzard ] }
FROM NAMED STREAM <http://www.cwi.nl/SRBench/observations>
           [NOW - 3 HOURS SLIDE 10 MINUTES]
WHERE {
  { SELECT ?sensor
    WHERE {
      ?sensor om-owl:generatedObservation [a weather:SnowfallObservation] ;
              om-owl:generatedObservation ?o1 ;
              om-owl:generatedObservation ?o2 .
      ?o1 a weather:TemperatureObservation ;
          om-owl:observedProperty weather:_AirTemperature ;
          om-owl:result [om-owl:value ?temperature] .
      ?o2 a weather:WindObservation ;
          om-owl:observedProperty weather:_WindSpeed ;
          om-owl:result [om-owl:value ?windSpeed] .
    }
    GROUP BY ?sensor
    HAVING ( AVG(?temperature) < "32"^^xsd:float && # fahrenheit
             MIN(?windSpeed) > "40.0"^^xsd:float ) #milesPerHour
  }
}
```

Query Properties:

- Use the `CONSTRUCT` query form
- Use `GROUP BY`, `HAVING` and aggregation functions
- Use subquery
- Requires a join between three types of observations on the sensor ID.
- Requires computing overlapping windows

6.6 Q6 – Get the stations that have observed extremely low visibility in the last hour.

```
PREFIX om-owl: <http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#>
PREFIX weather: <http://knoesis.wright.edu/ssw/ont/weather.owl#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>

SELECT ?sensor
FROM NAMED STREAM <http://www.cwi.nl/SRBench/observations> [NOW - 1 HOURS]
WHERE {
  { ?observation om-owl:procedure ?sensor ;
                 a weather:VisibilityObservation ;
                 om-owl:result [om-owl:floatValue ?value ] .
    FILTER ( ?value < "10"^^xsd:float) # centimeters
  }
  UNION
  { ?observation om-owl:procedure ?sensor ;
                 a weather:RainfallObservation ;
                 om-owl:result [om-owl:floatValue ?value ] .
    FILTER ( ?value > "30"^^xsd:float) # centimeters
  }
  UNION
  { ?observation om-owl:procedure ?sensor ;
                 a weather:SnowfallObservation .
  }
}
```

Query Properties:

- Use `FILTER` constraints
- Use `UNION` to match alternative patterns

6.7 Q7 – Detect stations that are recently broken.

```
PREFIX om-owl: <http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#>

SELECT DSTREAM DISTINCT ?sensor
FROM NAMED STREAM <http://www.cwi.nl/SRBench/observations> [NOW - 1 HOURS]
WHERE {
  ?sensor om-owl:generatedObservation ?observation .
}
```

Query Properties:

- Use the `DISTINCT` solution modifier
- Use `DSTREAM`

6.8 Q8 – Get the daily minimal and maximal air temperature observed by the sensor at a given location.

```
PREFIX om-owl: <http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#>
PREFIX weather: <http://knoesis.wright.edu/ssw/ont/weather.owl#>
PREFIX wgs84_pos: <http://www.w3.org/2003/01/geo/wgs84_pos>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>

SELECT ( MIN(?temperature) AS ?minTemperature )
       ( MAX(?temperature) AS ?maxTemperature )
FROM NAMED STREAM <http://www.cwi.nl/SRBench/observations> [NOW - 24 HOURS]
FROM <http://www.cwi.nl/SRBench/sensors>
WHERE {
  ?sensor om-owl:processLocation ?sensorLocation ;
```

```
          om-owl:generatedObservation ?observation .
  ?sensorLocation wgs84_pos:alt "%Altitude%"^^xsd:float ;
          wgs84_pos:lat "%Latitude%"^^xsd:float ;
          wgs84_pos:long "%Longitude%"^^xsd:float .
  ?observation om-owl:observedProperty weather:_AirTemperature ;
          om-owl:result [ om-owl:floatValue ?temperature ] .
}
GROUP BY ?sensor
```

Query Properties:

- Use both streaming data and static sensor metadata
- Use expressions in the `SELECT` clause
- Use `GROUP BY` and aggregation functions
- Requires a join between the sensor metadata and the observations on the sensor ID

6.9 Q9 – Get the daily average wind force and wind direction observed by the sensor at a given location.

```
PREFIX om-owl: <http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#>
PREFIX weather: <http://knoesis.wright.edu/ssw/ont/weather.owl#>
PREFIX wgs84_pos: <http://www.w3.org/2003/01/geo/wgs84_pos>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>

SELECT ( IF(AVG(?windSpeed) < 1, 0,
          IF(AVG(?windSpeed) < 4, 1,
           IF(AVG(?windSpeed) < 8, 2,
            IF(AVG(?windSpeed) < 13, 3,
             IF(AVG(?windSpeed) < 18, 4,
              IF(AVG(?windSpeed) < 25, 5,
               IF(AVG(?windSpeed) < 31, 6,
                IF(AVG(?windSpeed) < 39, 7,
                 IF(AVG(?windSpeed) < 47, 8,
                  IF(AVG(?windSpeed) < 55, 9,
                   IF(AVG(?windSpeed) < 64, 10,
                    IF(AVG(?windSpeed) < 73, 11, 12) ))))))))))
         AS ?windForce )
        ( AVG(?windDirection) AS ?avgWindDirection )
FROM NAMED STREAM <http://www.cwi.nl/SRBench/observations> [NOW - 24 HOURS]
FROM <http://www.cwi.nl/SRBench/sensors>
WHERE {
  ?sensor om-owl:processLocation ?sensorLocation ;
          om-owl:generatedObservation ?o1 ;
          om-owl:generatedObservation ?o2 .
  ?sensorLocation wgs84_pos:alt "%Altitude%"^^xsd:float ;
          wgs84_pos:lat "%Latitude%"^^xsd:float ;
          wgs84_pos:long "%Longitude%"^^xsd:float .
  ?o1 om-owl:observedProperty weather:_WindSpeed ;
      om-owl:result [ om-owl:floatValue ?windSpeed ] .
  ?o2 om-owl:observedProperty weather:_WindDirection ;
      om-owl:result [ om-owl:floatValue ?windDirection ] .
}
GROUP BY ?sensor
```

Query Properties:

- Use both streaming data and static sensor metadata
- Use expressions in the `SELECT` clause
- Use `IF` functions to map the resulting wind speed to the Beaufort Wind Force Scale.

- Use `GROUP BY` and aggregation functions
- Requires a join between the sensor metadata and the observations on the sensor ID

6.10 Q10 – Get the locations where a heavy snowfall has been observed in the last day.

```
PREFIX om-owl: <http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#>
PREFIX weather: <http://knoesis.wright.edu/ssw/ont/weather.owl#>
PREFIX wgs84_pos: <http://www.w3.org/2003/01/geo/wgs84_pos>

SELECT DISTINCT ?lat ?long ?alt
FROM NAMED STREAM <http://www.cwi.nl/SRBench/observations> [NOW - 24 HOURS]
FROM <http://www.cwi.nl/SRBench/sensors>
WHERE {
  ?sensor om-owl:generatedObservation [a weather:SnowfallObservation] .
  ?sensor om-owl:processLocation ?sensorLocation .
  ?sensorLocation wgs84_pos:alt ?alt ;
                  wgs84_pos:lat ?lat ;
                  wgs84_pos:long ?long .
}
```

Query Properties:

- Use the `DISTINCT` solution modifier
- Use both streaming data and static sensor metadata
- Requires a join between the sensor metadata and the observations on the sensor ID

6.11 Q11 – Detecting if a station has produced significantly different measurements than its neighbouring stations

```
PREFIX om-owl: <http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#>
PREFIX weather: <http://knoesis.wright.edu/ssw/ont/weather.owl#>
PREFIX wgs84_pos: <http://www.w3.org/2003/01/geo/wgs84_pos>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>

SELECT DISTINCT ?sensor
FROM NAMED STREAM <http://www.cwi.nl/SRBench/observations> [NOW - 1 HOURS]
FROM <http://www.cwi.nl/SRBench/sensors>
WHERE {
  ?sensor om-owl:generatedObservation ?observation ;
          om-owl:hasLocatedNearRel [om-owl:hasLocation ?nearbyLocation] .
  ?observation a ?observationType ;
               om-owl:observedProperty ?observationProperty ;
               om-owl:result [ om-owl:floatValue ?value ] .
  { SELECT AVG(?value2) AS ?avgValue
    WHERE {
      ?sensor2 om-owl:generatedObservation ?observation2 ;
               om-owl:hasLocatedNearRel [om-owl:hasLocation ?nearbyLcation2] .
      FILTER ( sameTerm(?nearbyLocation, ?nearbyLocation2) )
      ?observation2 a ?observationType ;
                    om-owl:observedProperty ?observationProperty ;
                    om-owl:result [ om-owl:floatValue ?value2 ] .
    }
  }
  FILTER ( ABS(?value - ?avgValue) / ?avgValue > "0.10"^^xsd:float)
}
```

Query Properties:

- Use both streaming data and static sensor metadata

- Use the `DISTINCT` solution modifier
- Use subquery
- Use the `sameTerm`, `ABS` and aggregation functions
- Use `FILTER` constraints
- Reasoning over the `om-owl:hasLocatedNearRel` property to find neighbouring sensor stations.
- Requires joins between the sensor metadata and the observations on the sensor ID, and between different observations on the observation type and observed property.

6.12 Q12 – Get the hourly average air temperature and humidity of large cities.

```
PREFIX gn: <http://www.geonames.org/ontology#>
PREFIX om-owl: <http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#>
PREFIX weather: <http://knoesis.wright.edu/ssw/ont/weather.owl#>
PREFIX wgs84_pos: <http://www.w3.org/2003/01/geo/wgs84_pos>

SELECT ?name ( AVG(?temperature) AS ?avgTemperature )
       ( AVG(?humidity) AS ?avgHumidity )
FROM NAMED STREAM <http://www.cwi.nl/SRBench/observations>
           [NOW - 1 HOURS SLIDE 1 HOURS]
FROM <http://www.cwi.nl/SRBench/sensors>
FROM <http://www.cwi.nl/SRBench/geonames>
WHERE {
  ?sensor om-owl:generatedObservation ?temperatureObservation;
          om-owl:generatedObservation ?humidityObservation;
          om-owl:hasLocatedNearRel [ om-owl:hasLocation ?nearbyLocation ] .
  ?temperatureObservation om-owl:observedProperty weather:_AirTemperature ;
                          om-owl:result [ om-owl:floatValue ?temperature ] .
  ?humidityObservation om-owl:observedProperty weather:_RelativeHumidity ;
                       om-owl:result [ om-owl:floatValue ?humidity ] .
  { SELECT ?name
    WHERE {
      ?nearbyLocation gn:featureClass ?featureClass ;
                      gn:name | gn:officialName ?name ;
                      gn:population ?population .
      FILTER ( ?population > 15000 && REGEX(?featureClass, "P" , "i") )
    }
  }
  UNION
  { SELECT ?name
    WHERE {
      ?nearbyLocation gn:parentFeature+ ?parentFeature .
      ?parentFeature gn:featureClass ?parentClass ;
                     gn:name | gn:officialName ?name ;
                     gn:population ?parentPopulation .
      FILTER ( ?parentPopulation > 15000 && REGEX(?parentClass, "P" , "i") )
    }
  }
}
GROUP BY ?name
```

Query Properties:

- Interlink the LSD dataset and the GeoNames dataset
- Requires computing disjoint windows
- Use expressions in the `SELECT` clause
- Use `GROUP BY` and aggregation functions

- Requires joins between the sensor metadata and the observations on the sensor ID, and between a sensor and a geographic feature on their location
- Use subqueries
- Use `FILTER` constraints and the `REGEX` function
- Use `UNION` to match alternative patterns
- Use Property Path expression with alternatives and with an arbitrary length path

6.13 Q13 – Get the shores in Florida, US where a strong wind, i.e., the wind force is between 6 and 9, has been observed in the last hour.

```
PREFIX gn: <http://www.geonames.org/ontology#>
PREFIX om-owl: <http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#>
PREFIX weather: <http://knoesis.wright.edu/ssw/ont/weather.owl#>
PREFIX wgs84_pos: <http://www.w3.org/2003/01/geo/wgs84_pos>

SELECT ?shoreName ?lat ?long
      ( IF(AVG(?windSpeed) < 31, 6,
         IF(AVG(?windSpeed) < 39, 7, IF(AVG(?windSpeed) < 47, 8, 9)))
        AS ?windForce )
FROM NAMED STREAM <http://www.cwi.nl/SRBench/observations> [NOW - 1 HOURS]
FROM <http://www.cwi.nl/SRBench/sensors>
FROM <http://www.cwi.nl/SRBench/geonames>
WHERE {
  ?shore gn:featureClass ?shoreClass ;
         wgs84_pos:lat ?lat ;
         wgs84_pos:long ?long ;
         gn:name|gn:officialName ?shoreName ;
         gn:parentFeature+ ?florida .
  ?florida gn:name|gn:officialName ?floridaName .
  FILTER ( ( REGEX(?shoreClass, "L.CST" , "i") || # coast
             REGEX(?shoreClass, "T.BCH" , "i") || # beach
             REGEX(?shoreClass, "T.SHOR" , "i") ) && # shore
           REGEX(?floridaName, "Florida", "i") )
  ?sensor om-owl:generatedObservation ?observation;
          om-owl:hasLocatedNearRel [ om-owl:hasLocation ?shore ] .
  ?observation om-owl:observedProperty weather:_WindSpeed ;
               om-owl:result [ om-owl:floatValue ?windSpeed ] .
  FILTER ( 25 <= ?windSpeed || ?windSpeed <= 54 ) # milesPerHour
}
GROUP BY ?shoreName ?lat ?long
```

Query Properties:

- Interlink the LSD dataset and the GeoNames dataset
- Use expressions in the `SELECT` clause
- Use `IF` functions to map the resulting wind speed to the Beaufort Wind Force Scale.
- Use `GROUP BY` and aggregation functions
- Requires joins between the sensor metadata and the observations on the sensor ID, and between a sensor and a geographic feature on their location
- Use `FILTER` constraints and the `REGEX` function
- Use Property Path expressions with alternatives and with an arbitrary length path

6.14 Q14 – Get the airport(s) located in the same city as the sensor that has observed extremely low visibility in the last hour.

```
PREFIX gn: <http://www.geonames.org/ontology#>
PREFIX om-owl: <http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#>
PREFIX weather: <http://knoesis.wright.edu/ssw/ont/weather.owl#>
PREFIX wgs84_pos: <http://www.w3.org/2003/01/geo/wgs84_pos>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>

SELECT DISTINCT ?airportName ?lat ?long
FROM NAMED STREAM <http://www.cwi.nl/SRBench/observations> [NOW - 1 HOURS]
FROM <http://www.cwi.nl/SRBench/sensors>
FROM <http://www.cwi.nl/SRBench/geonames>
WHERE {
  ?airport gn:featureClass ?airportClass ;
           wgs84_pos:lat ?lat ;
           wgs84_pos:long ?long ;
           gn:name|gn:officialName ?airportName ;
           gn:parentFeature+ ?city .
  ?city gn:featureClass ?cityClass .
  FILTER ( REGEX(?airportClass, "S.AIRP" , "i") &&
           REGEX(?cityClass, "P" , "i") )
  ?sensor om-owl:generatedObservation ?observation;
          om-owl:hasLocatedNearRel [ om-owl:hasLocation ?city ] .
  { ?observation om-owl:procedure ?sensor ;
                 a weather:VisibilityObservation ;
                 om-owl:result [om-owl:floatValue ?value ] .
    FILTER ( ?value < "10"^^xsd:float) # centimeters
  }
  UNION
  { ?observation om-owl:procedure ?sensor ;
                 a weather:RainfallObservation ;
                 om-owl:result [om-owl:floatValue ?value ] .
    FILTER ( ?value > "30"^^xsd:float) # centimeters
  }
  UNION
  { ?observation om-owl:procedure ?sensor ;
                 a weather:SnowfallObservation .
  }
}
```

Query Properties:

- Interlink the LSD dataset and the GeoNames dataset
- Use the `DISTINCT` solution modifier
- Requires joins between the sensor metadata and the observations on the sensor ID, and between a sensor and a geographic feature on their location
- Use `FILTER` constraints and the `REGEX` function
- Use `UNION` to match alternative patterns
- Use Property Path expressions with alternatives and with an arbitrary length path

6.15 Q15 – Get the locations where the wind speed in the last hour is higher than a known hurricane.

```
PREFIX dbpprop: <http://dbpedia.org/property/>
PREFIX om-owl: <http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
```

```
PREFIX weather: <http://knoesis.wright.edu/ssw/ont/weather.owl#>
PREFIX wgs84_pos: <http://www.w3.org/2003/01/geo/wgs84_pos>
PREFIX yago: <http://dbpedia.org/class/yago/>

SELECT ?lat ?long ?alt( AVG(?windSpeed) AS ?avgWindSpeed )
FROM NAMED STREAM <http://www.cwi.nl/SRBench/observations>
           [NOW - 1 HOURS SLIDE 1 HOURS]
FROM <http://www.cwi.nl/SRBench/sensors>
FROM <http://www.cwi.nl/SRBench/dbpedia>
WHERE {
  ?observation a weather:WindspeedObservation ;
               om-owl:procedure ?sensor ;
               om-owl:result [ om-owl:floatValue ?windSpeed ] .
  ?sensor om-owl:processLocation ?sensorLocation .
  ?sensorLocation wgs84_pos:alt ?alt ;
                  wgs84_pos:lat ?lat ;
                  wgs84_pos:long ?long .
  ?hurricane rdf:type/rdfs:subClassOf* yago:Hurricane111467018 ;
             dbpprop:1MinWinds ?hurricaneWindSpeed.
  FILTER(?windSpeed > ?hurricaneWindSpeed)
}
GROUP BY ?lat ?long ?alt
```

Query Properties:

- Interlink the LSD dataset and the DBpedia dataset
- Use expressions in the `SELECT` clause
- Use `GROUP BY` and aggregation functions
- Use `FILTER` constraints
- Requires a join between the sensor metadata and the observations on the sensor ID
- Use Property Path expression with an arbitrary length path, to reason over all instances of the class `yago:Hurricane111467018` and its subclasses

6.16 Q16 – Get the heritage sites that are threatened by a hurricane.

```
PREFIX category: <http://dbpedia.org/resource/Category:>
PREFIX dcterms: <http://purl.org/dc/terms/>
PREFIX om-owl: <http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#>
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX skos: <http://www.w3.org/2004/02/skos/core#>
PREFIX weather: <http://knoesis.wright.edu/ssw/ont/weather.owl#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>

SELECT ?heritage
FROM NAMED STREAM <http://www.cwi.nl/SRBench/observations>
FROM <http://www.cwi.nl/SRBench/sensors>
FROM <http://www.cwi.nl/SRBench/geonames>
FROM <http://www.cwi.nl/SRBench/dbpedia>
WHERE {
  ?observation a weather:WindspeedObservation ;
               om-owl:procedure ?sensor ;
               om-owl:result [ om-owl:floatValue ?windSpeed ] .
  FILTER ( ?windSpeed >= "74"^^xsd:float ) #milesPerHour
  ?sensor om-owl:hasLocatedNearRel [om-owl:hasLocation ?nearbyLocation] .
  ?heritage owl:sameAs ?nearbyLocation ;
            dcterms:subject ?category .
  ?category skos:broader* category: World_Heritage_Sites .
}
```

Query Properties:

- Interlink the LSD dataset and the DBpedia dataset
- Use `FILTER` constraints
- Use `owl:sameAs` reasoning
- Requires joins between the sensor metadata and the observations on the sensor ID, and between the sensor data and the DBpedia data on the `?nearbyLocation`.
- Use Property Path expression with an arbitrary length path

6.17 Q17 – Estimate the damage where a hurricane has been observed.

```
PREFIX dbpprop: <http://dbpedia.org/property/>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX gn: <http://www.geonames.org/ontology#>
PREFIX om-owl: <http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX weather: <http://knoesis.wright.edu/ssw/ont/weather.owl#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX yago: <http://dbpedia.org/class/yago/>

SELECT ?damage
FROM NAMED STREAM <http://www.cwi.nl/SRBench/observations>
FROM <http://www.cwi.nl/SRBench/sensors>
FROM <http://www.cwi.nl/SRBench/geonames>
FROM <http://www.cwi.nl/SRBench/dbpedia>
WHERE {
  ?observation a weather:WindspeedObservation ;
               om-owl:procedure ?sensor ;
               om-owl:result [ om-owl:floatValue ?windSpeed ] .
  FILTER ( ?windSpeed >= "74"^^xsd:float ) #milesPerHour
  ?sensor om-owl:hasLocatedNearRel [om-owl:hasLocation ?nearbyLocation] .
  ?hurricane dbpprop:areas [ foaf:name ?areaName ] ;
             rdf:type/rdfs:subClassOf* yago:Hurricane111467018 ;
             dbpprop:damages ?damage .
  ?nearbyLocation gn:parentFeature* ?area .
  ?area gn:name|gn:officialName ?areaName .
}
```

Query Properties:

- Interlink the LSD, DBpedia and GeoNames datasets.
- Use `FILTER` constraints
- Requires joins between the sensor metadata and the observations on the sensor ID, between the sensor data and the GeoNames data on the `?nearbyLocation`, and between the GeoNames data and DBpedia data on `?areaName`
- Use Property Path expressions with alternatives and with an arbitrary length path

6.18 Discussion

An overview of RDF/SPARQL features used by each query is shown in Table 2. In general, the functionality supported by $SPARQL_{Stream}$ is fairly complete. At the language level, it is able to express all benchmark queries easily and concisely. At the query processing level, some missing features have been discovered. During the evaluation, UPM and CWI have closely collaborated to improve the quality of the existing code and extend the code to support the missing features. The development on the new features will continue under the collaboration of the two partners.

First of all, errors caused by out-of-date software code, in the *Stream-translator*, which translate a query written in $SPARQL_{Stream}$ into a query using the language implemented by the underlying streaming data processor, have been corrected.

The (improved) $SPARQL_{Stream}$ engine now supports all language syntax required by the benchmark, but running queries containing complex $SPARQL_{Stream}$ language features, for instance, `OPTIONAL`, complex `FILTER`s like those with `REGEX`, needs more development efforts.

Because the aggregation functions AVG(), MIN(), MAX() have been introduced only in SPARQL 1.1, implementation of these functions haven't been finished. Some preliminary code to support GROUP BY, HAVING and ASK have been added, since the evaluation. The development of these features will continue.

So far, the development of the $SPARQL_{Stream}$ engine has concentrated on supporting streaming data, but not on supporting both streaming and static data. Furthermore, the `CONSTRUCT` queries used to only accept static RDF data, but no streaming data. As a result of the evaluation, code has been added to support both streaming and static RDF data in all kinds of queries and will be further developed.

The query Q5 uses a subquery as a workaround to overcome the problem that the use of the `GROUP BY` variable `?sensor` in the `CONSTRUCT` clause can otherwise not be detected by the parser. This problem is cause by that in the ARQ framework[51], `CONSTRUCT` queries are treated like `SELECT *` queries, which fails to detect that the `GROUP BY` variable `?sensor` is actually use in the `CONSTRUCT` clause. Since this is a problem in third-party software, we cannot estimate if and when it will be fixed.

Finally, we have noticed during the evaluation that the window-to-stream operators, such as the `DSTREAM` operator, although rarely used, but can be very useful in several important use cases that are specific for streaming applications. For instance, the `ISTREAM` and `DSTREAM` can be used to detect sudden changes in the availability of sensors. So far, support for these operators has been ignored because some details of their semantics are still unclear. However, the use case of `DSTREAM` in the benchmark motivates us to consider implementing these operators.

Table 2: An overview of RDF/SPARQL features used by each query

	DISTINCT	FILTER	Regular expression	Expanded functions	OPTIONAL	SELECT with expressions	UNION	IF	Subquery	GROUP BY	Aggregation	HAVING	ASK	CONSTRUCT	Property path	Subclasses	Subproperties	owl:sameAs
Q1	✓																	
Q2	✓	✓	✓	✓	✓										✓			
Q3										✓	✓	✓	✓				✓	
Q4		✓	✓	✓		✓				✓	✓							
Q5									✓	✓	✓	✓		✓				
Q6		✓					✓											
Q7	✓																	
Q8						✓				✓	✓							
Q9						✓		✓		✓	✓							
Q10	✓																	
Q11	✓	✓		✓					✓	✓	✓							
Q12		✓	✓			✓	✓		✓	✓	✓				✓			
Q13		✓	✓			✓				✓	✓				✓			
Q14	✓	✓	✓				✓								✓			
Q15		✓				✓				✓	✓				✓	✓		
Q16		✓													✓			✓
Q17		✓													✓	✓		

[51] See: http://jena.sourceforge.net/documentation.html

7 Conclusion and Future Work

In this deliverable, we have presented version V1.0 of SRBench, the first general purpose Streaming RDF Benchmark, which has been designed from scratch to assess the streaming RDF engines.

The benchmark has been designed based on an extensive study of the state-of-the-art techniques in both the data stream management systems and the streaming RDF processing engines. This ensures that we capture all important aspects of streaming RDF processing in the benchmark.

Motivated by the study of (Duan et al., 2011), we have chosen on purpose to not generate synthetic data for the benchmark, but use real-world datasets instead. Thus, SRBench uses three real-world datasets, i.e., the LinkedSensorData dataset17, the GeoNames RDF dataset15 and the DBpedia dataset16. The LinkedSensorData dataset contains streaming RDF data collected from US weather stations. It is the first and largest sensor dataset in the Linked Open Data cloud and the CKAN portal. The LinkedSensorData links with the GeoNames RDF dataset through the `hasLocationNearRel` property, which points to a location defined by GeoNames to which a sensor is located close. The GeoNames dataset is linked with the DBpedia dataset through the `owl:sameAs` property, which denotes that two subjects described by the two datasets are about the same place.

The goal of SRBench V1.0 is to evaluate the functional completeness of a streaming RDF engine. The benchmark contains a concise, yet comprehensive set of queries which covers the major aspects of streaming SPARQL query processing, ranging from simple pattern matching queries to queries with complex reasoning tasks. The main advantages of applying Semantic Web technologies on streaming data include providing better search facilities by adding semantics to the data, reasoning through ontologies, and integration with other data sets. The ability of a streaming RDF engine to process these distinctive features is accessed by the benchmark with queries that apply reasoning not only over the streaming sensor data, but also over the metadata and even other data sets in the Linked Open Data (LOD) cloud.

Finally, we have complemented our work on SRBench with a functional evaluation of the benchmark on the $\text{SPARQL}_{\text{Stream}}$ query-processing engine developed by the PlanetData partner UPM. The evaluation shows that the functionality supported by $\text{SPARQL}_{\text{Stream}}$ is fairly complete. At the language level, it is able to express all benchmark queries easily and concisely. At the query processing level, some missing features have been discovered, for all of which preliminary code has been added for further development.

The work on SRBench is our first step in developing and benchmarking streaming RDF engines. A natural next step is to run performance and scalability evaluation on different streaming RDF engines and continuously improve the benchmark. We have started working on MonetDB/DataCell streaming engine, which will be reported in deliverable D1.7 in month 42. During the functional evaluation, we have discovered that the current code base of $\text{SPARQL}_{\text{Stream}}$ can be reasonably easily extended to operate on top of MonetDB/DataCell. Thus the collaboration between UPM and CWI will be continued.

References

Abadi, D. J., Ahmad, Y., Balazinska, M., Çetintemel, U., Cherniack, M., Hwang, J.-H., Lindner, W., Maskey, A. S., Rasin, A., Ryvkina, E., Tatbul, N., Xing, Y., and Zdonik, S. (2005). *The Design of the Borealis Stream Processing Engine*. In CIDR.

Abadi, D. J., Carney, D., Çetintemel, U., Cherniack, M., Convey, C., Erwin, C., Galvez, E. F., Hatoun, M., Maskey, A., Rasin, A., Singer, A., Stonebraker, M., Tatbul, N., Xing, Y., Yan, R., and Zdonik, S. B. (2003a). *Aurora: A Data Stream Management System*. In SIGMOD Conference, page 666.

Abadi, D. J., Carney, D., Çetintemel, U., Cherniack, M., Convey, C., Lee, S., Stonebraker, M., Tatbul, N., and Zdonik, S. (2003b). *Aurora: a new model and architecture for data stream management*. The VLDB Journal, 12:120– 139.

Aberer, K., Hauswirth, M., and Salehi, A. (2006a). *A middleware for fast and flexible sensor network deployment*. In Very Large Data Bases (VLDB), Seoul, Korea, 2006.

Aberer, K., Hauswirth, M., and Salehi, A. (2006b). *The Global Sensor Networks middleware for efficient and flexible deployment and interconnection of sensor networks*. In Technical Report, LSIR-2006-006, Ecole Polytechnique Fédérale de Lausanne (EPFL), Lausanne, Switzerland.

Ali, M. H., Gerea, C., Raman, B. S., Sezgin, B., Tarnavski, T., Verona, T., Wang, P., Zabback, P., Kirilov, A., Ananthanarayan, A., Lu, M., Raizman, A., Krishnan, R., Schindlauer, R., Grabs, T., Bjeletich, S., Chandramouli, B., Goldstein, J., Bhat, S., Li, Y., Nicola, V. D., Wang, X., Maier, D., Santos, I., Nano, O., and Grell, S. (2009). *Microsoft CEP server and online behavioral targeting*. PVLDB, 2(2):1558–1561.

Anicic, D., Fodor, P., Rudolph, S., Stühmer, N., Stojanovic, N. and Studer, N. (2011a). *Etalis: Rule-based reasoning in event processing*. In Reasoning in Event-based Distributed Systems, Studies in Computational Intelligence series. LNCS, Springer Verlag.

Anicic, D., Fodor, P., Rudolph, S., and Stojanovic, N. (2011b). *EP-SPARQL: a unified language for event processing and stream reasoning*. In WWW ’11, pages 635–644, 2011.

Arasu, A., Cherniack, M., Galvez, E., Maier, D., Maskey, A. S., Ryvkina, E., Stonebraker, M., and Tibbetts, R. (2004). *Linear Road: A Stream Data Management Benchmark*. In Proc. Of the 30th VLDB Conference, pages 480–491, Toronto, Canada.

Arasu, A., Babcock, B., Babu, S., Datar, M., Ito, K., Nishizawa, I., Rosenstein, J., and Widom, J. (2003a). *STREAM: The Stanford Stream Data Manager*. In Proc. of the ACM SIGMOD Int’l. Conf. on Management of Data.

Arasu, A., Babu, S., and Widom, J. (2003b). *CQL: A Language for Continuous Queries over Streams and Relations*. In DBPL2003, pages 1–19.

Avnur, R. and Hellerstein, J. M. (2000). *Eddies: continuously adaptive query processing*. SIGMOD Rec., 29:261–272.

Babcock, B., Babu, S., Datar, M., Motwani, R., and Thomas, D. (2004). *Operator Scheduling in Data Stream Systems*. The VLDB Journal, 13(4):333–353.

Babcock, B., Babu, S., Motwani, R., and Datar, M. (2003). *Chain: operator scheduling for memory minimization in data stream systems*. In Proceedings of the 2003 ACM SIGMOD international conference on Management of data, SIGMOD ’03, pages 253–264, New York, NY, USA. ACM.

Babu, S. and Widom, J. (2004). *StreaMon: an adaptive engine for stream query processing*. In Proceedings of the 2004 ACM SIGMOD international conference on Management of data, SIGMOD ’04, pages 931–932, New York, NY, USA. ACM.

Balakrishnan, H., Balazinska, M., Carney, D., Çetintemel, U., Cherniack, M., Convey, C., Galvez, E., Salz, J., Stonebraker, M., Tatbul, N., Tibbetts, R., and Zdonik, S. (2004). *Retrospective on Aurora*. The VLDB Journal, 13(4):370–383.

Balazinska, M., Deshpande, A., Franklin, M. J., Gibbons, P. B., Gray, J., Hansen, M., Liebhold, M., Nath, S., Szalay, A. and Tao, V. (2007). *Data Management in the Worldwide Sensor Web*. IEEE Pervasive Computing, 6(2):30–40, 2007.

Barrasa, J., Corcho, O., and Gómez-Pérez, A. (2004). *R2O, an extensible and semantically based database-to-ontology mapping language*. In: SWDB2004, pages1069–1070.

Barbieri, D.F., Braga, D., Ceri, S., Della Valle, E., and Grossniklaus, M. (2010a). *Querying RDF Streams with C-SPARQL*. In SIGMOD Record, 39(1):20–26, March 2010.

Barbieri, D.F., Braga, D., Ceri, S., and Grossniklaus, M. (2010b). *An execution environment for C-SPARQL queries*. In EDBT 2010, pages 441–452, Lausanne, Switzerland.

Barbieri, D.F., Braga, D., Ceri, S., Della Valle, E., and Grossniklaus, M. (2009). *C-SPARQL: SPARQL for Continuous Querying*. In WWW 2009, pages 1061–1062, April 2009, Madrid, Spain.

Bolles, A., Grawunder, M., and Jacobi, J. (2008). *Streaming SPARQL – extending SPARQL to process data streams*. In ESWC 08, pages 448–462.

Bouillet, E., Feblowitz, M., Liu, Z., Ranganathan, A., Riabov, A., and Ye, F. (2007). *A semantics-based middleware for utilizing heterogeneous sensor networks*. In DCOSS'07, pages 174–188, 2007.

Brenninkmeijer, C.Y., Galpin, I., Fernandes, A.A., and Paton, N.W. (2008). *A semantics for a query language over sensors, streams and relations*. In: BNCOD '08, pages 87–99.

Calbimonte, J.-P., Corcho, O., and Gray, A. J. G. (2010). *Enabling Ontology-based Access to Streaming Data Sources*. In 9th International Semantic Web Conference (ISWC2010).

Carney, D., Çetintemel, U., Cherniack, M., Convey, C., Lee, S., Seidman, G., Stonebraker, M., Tatbul, N., and Zdonik, S. (2002). *Monitoring streams: a new class of data management applications*. In Proceedings of the 28th international conference on Very Large Data Bases, VLDB '02, pages 215–226. VLDB Endowment.

Carroll, J.J., Dickinson, I., Dollin, C., Reynolds, D., Seaborne, A., and Wilkinson, K. (2004). *Jena: implementing the semantic web recommendations*. In: WWW Alt. '04: Proceedings of the 13th international World Wide Web conference on Alternate track papers & posters, New York, NY, USA, ACM Press, pages 74–83.

Chandrasekaran, S., Cooper, O., Deshpande, A., Franklin, M. J., Hellerstein, J. M., Hong, W., Krishnamurthy, S., Madden, S., Raman, V., Reiss, F., and Shah, M. A. (2003). *TelegraphCQ: Continuous Dataflow Processing for an Uncertain World*. In Proc. of the Int'l Conf. on Innovative Database Systems Research (CIDR).

Chandrasekaran, S. and Franklin, M. J. (2002). *Streaming queries over streaming data*. In Proceedings of the 28th international conference on Very Large Data Bases, VLDB '02, pages 203–214. VLDB Endowment.

Chen, J., Dewitt, D. J., Tian, F., and Wang, Y. (2000). *NiagaraCQ: A Scalable Continuous Query System for Internet Databases*. In Proc. of the ACM SIGMOD Int'l. Conf. on Management of Data.

Chen, Q. M. and Hsu, M. C. (2010). *Experience in Extending Query Engine for Continuous Analytics*. Technical Report TR-44, HP Laboratories.

Corcho, O. and García-Castro, R. (2010). *Five challenges for the semantic sensor web*. In Semantic Web, 1(1):121–125.

Corcho, O., Priyatna, F., Fortuna, C., Grobelnik, M., Calbimonte, J.-P., García-Silva, A., Jeung, H. Y., Novak, B., and Moraru, A.(2011). *D1.1 Characterisation mechanisms for unknown data sources*. In EU Project PlanetData (FP7-257641), Deliverable 1.1. http://www.planet-data.eu/sites/default/files/pr-material/deliverables/D1.1_Characterisation_mechanics_for_unknown_data_sources.pdf

Cranor, C. D., Johnson, T., Spatscheck, O., and Shkapenyuk, V. (2003). *Gigascope: A Stream Database for Network Applications*. In Proc. of the ACM SIGMOD Int'l. Conf. on Management of Data.

Della Valle, E., Ceri, S., van Harmelen, F., and Fensel, D. (2009). *It's a Streaming World! Reasoning upon Rapidly Changing Information*. In IEEE Intelligent Systems, 24(6): 83-89, November/December 2009.

Duan, S., Kementsietsidis, A., Srinivas, K., and Udrea, O. (2011). *Apples and Oranges: A Comparison of RDF Benchmarks and Real RDF Datasets*. In SIGMOD'11, June 2011, Athens, Greece.

Franklin, M. J., Krishnamurthy, S., Conway, N., Li, A., Russakovsky, A., and Thombre, N. (2009). *Continuous analytics: Rethinking query processing in a network-effect world*. In Proc. of the Int'l Conf. on Innovative Database Systems Research (CIDR).

Galpin, I., Brenninkmeijer, C.Y., Jabeen, F., Fernandes, A.A., and Paton, N.W. (2009). *Comprehensive optimization of declarative sensor network queries*. In: SSDBM 2009, pages 339–360.

Gedik, B., Andrade, H., Wu, K.-L., Yu, P. S., and Doo, M. (2008). *Spade: the system s declarative stream processing engine*. In Proceedings of the 2008 ACM SIGMOD international conference on Management of data, SIGMOD '08, pages 1123–1134, New York, NY, USA. ACM.

Guo, Y., Pan, Z., and Heflin, J. (2004). *An evaluation of knowledge base systems for large OWL datasets*. In: The Semantic Web - ISWC 2004, Springer-Berlin/Heidelberg, pages 274–288.

Groppe, S., Groppe, J., Kukulenz, D., and Linnemann, V. (2007). *A SPARQL Engine for Streaming RDF Data*. In Proc. of the 13th International IEEE Conference on Signal-Image Technologies and Internet-Based System (SITIS07).

Groppe., S. (2011). *Data Management and Query Processing in Semantic Web Databases*. ISBN 978-3-642-19356-9, Springer-Verlag Berlin Heidelberg, 2011.

Harris, S. and Seaborne, A. (2012). *SPARQL 1.1 Query Language*. W3C Working Draft 05 January 2012, World Wide Web Consortium, http://www.w3.org/TR/sparql11-query/

Hoeksema, J. (2011). *A Parallel RDF Stream Reasoner and C-SPARQL Processor Using the S4 Framework*. Master thesis, VU University, Amsterdam, the Netherlands, Oct. 2011.

Ivanova, M., Kersten, M. L., Nes, N. and Goncalves, R. (2010). *An Architecture for Recycling Intermediates in a Column-store*. In Transactions on Database Systems (TODS), 35(4), 2010.

Ivanova, M., Kersten, M. L., Nes, N. and Goncalves, R. (2009). *An Architecture for Recycling Intermediates in a Column-store*. In Proc. of the ACM SIGMOD Conference, Providence, RI, USA, June 29- July 2, 2009.

Jain, P., Hitzler, P., Yeh, P.Z., Verma, K., and Sheth, A.P. (2009). *Linked Data is Merely More Data*. In Linked Data Meets Artificial Intelligence.

Kossmann, D. (2000). *The state of the art in distributed query processing*. ACM Comput. Surv. 32(4): 422–469.

Le-Phuoc, D., Dao-Tran, M., Parreira, J. X., and Hauswirth, M. (2011a). *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, pages 370–388, Bonn, Germany.

Le-Phuoc, D., Parreira, J. X., and Hauswirth, M. (2011b). *A Native and Adaptive Approach for Unified Processing of Linked Streams and Linked Data*. Technical Report DERI-TR-2011-07-06, DERI, IDA Business Park, Lower Dangan, Galway, Ireland, July 2011.

Le-Phuoc, D., Parreira, J. X., Hausenblas, M., and Hauswirth, M. (2010). *Continuous query optimization and evaluation over unified linked stream data and linked open data*. Technical Report DERI-TR-2010-09-27, DERI, IDA Business Park, Lower Dangan, Galway, Ireland, Sep. 2010.

Le-Phuoc, D. and Hauswirth, M. (2009). *Linked open data in sensor data mashups*. In Proc. Semantic Sensor Networks, page 1.

Liarou, E., Goncalves, R., and Idreos, S. (2009). *Exploiting the power of relational databases for efficient stream processing*. In Proc. of the Intl. Conf. on Extending Database Technology (EDBT).

Madden, S., Shah, M. A., Hellerstein, M. J., and Raman, V. (2002). *Continuously Adaptive Continuous Queries over Streams*. In Proc. of the ACM SIGMOD Int'l. Conf. on Management of Data.

Motwani, R., Widom, J., Arasu, A., Babcock, B., Babu, S., Datar, M., Manku, G., Olston, C., Rosenstein, J., and Varma, R. (2002). *Query processing, resource management, and approximation in a data stream management system*. Technical Report 2002-41, Stanford InfoLab.

Motwani, R., Widom, J., Arasu, A., Babcock, B., Babu, S., Datar, M., Manku, G. S., Olston, C., Rosenstein, J., and Varma, R. (2003). *Query processing, approximation, and resource management in a data stream management system*. In CIDR.

Pérez, J., Arenas, M., Gutierrez, C. (2009). *Semantics and complexity of SPARQL*. ACM Trans. Database Syst. 34(3):1–45.

Prud'hommeaux, E. and Seaborne, A. (2008). *SPARQL Query Language for RDF*. W3C Recommendation 15 January 2008, World Wide Web Consortium, http://www.w3.org/TR/rdf-sparql-query/.

Rapoza, J. (2006). *SPARQL Will Make the Web Shine*. In eWeek.com, 02 May, 2006, http://www.eweek.com/c/a/Application-Development/SPARQL-Will-Make-the-Web-Shine/.

Segaran, T., Evans, C., Taylor, J. (2009). *Programming the Semantic Web*. O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472. p. 84. ISBN 978-0-596-15381-6.

Ruyter, B. de and Pelgrim, E. (2007). *Ambient assisted-living research in carelab*. In Interactions, 14(4):30–33, ISSN 1072-5520,July, 2007.

Sbz, S. Z., Zdonik, S., Stonebraker, M., Cherniack, M., Etintemel, U. C., Balazinska, M., and Balakrishnan, H. (2003). *The Aurora and Medusa projects*. IEEE Data Engineering Bulletin, 26.

Sequeda, J. and Corcho, O. (2009). *Linked stream data: A position paper*. In Proc. Semantic Sensor Networks, page 148.

Sheth, A. P., Henson, C. A., and Sahoo, S. S. (2008). *Semantic Sensor Web*. IEEE Internet Computing, 12(4):78–83, 2008.

Tatbul, E. N. (2007). *Load shedding techniques for data stream management systems*. PhD thesis, Providence, RI, USA. AAI3272068.

Walavalkar, O., Joshi, A., Finin, T., and Yesha, Y. (2008). *Streaming Knowledge Bases*. In Proc. of the 4th International Workshop on Scalable Semantic Web Knowledge Base System (SSWS2008).

Whitehouse, K., Zhao, F., and Liu, J. (2006). *Semantic Streams: A Framework for Composable Semantic Interpretation of Sensor Data*. In European Workshop on Wireless Sensor Networks, pages 5–20. EWSN, 2006.

Winter, R. and Kostamaa, P. (2010). *Large scale data warehousing: Trends and observations*. In Proc. of the Int'l. Conf. on Database Engineering (ICDE).

Annex A The SPARQL$_{Stream}$ Streaming RDF Query Language

In this section we describe the SPARQL$_{Stream}$ query language, an extension to SPARQL for streaming RDF data, which has been inspired by previous proposals such as C-SPARQL (Barbieri et al., 2009; Barbieri et al., 2010a; Barbieri et al., 2010b) and SNEEql (Brenninkmeijer, et al., 2008). However, significant improvements have been made that correct the types supported and the semantics of windowing operations, which can be summarised as:

- Only windows defined in time are supported;
- The result of a window operation is a window of triples, not a stream, over which traditional operators can be applied, as such window-to-stream operators have been added; and
- The SPARQL 1.1 definition has been adopted for aggregates.

A.1 SPARQL$_{Stream}$ Syntax

Just as in C-SPARQL, an RDF stream is defined as a sequence of pairs (T_i,τ_i) where T_i is an RDF triple $\langle s_i,p_i,o_i\rangle$ and τ_i is a time stamp which comes from a monotonically non-decreasing sequence. An RDF stream is identified by an IRI, which provides the location of the data source. The IRI's identify virtual RDF streams since they are derived from the streaming data sources.

Window definitions are of the form "`FROM` *Start* `TO` *End* `[SLIDE]` `[`*Literal*`]`", where the *Start* and *End* are of the form `NOW` or `NOW` `-` *Literal*, where *Literal* represents some number of time unit (`DAYS`, `HOURS`, `MINUTES`, or `SECONDS`). The parser also accepts the non-plural form of the time units and is not case sensitive. The optional `SLIDE` indicates the gap between each successive window evaluation. Note that if the size of the slide is smaller than the range of the window, then the windows will overlap; if the size of the slide coincides with the size of the window, then every triple will appear in one and only one window; and if the slide is larger than the range of the window, then the windows sample the stream. Also note that the definition of a window can be completely in the past. This is useful for correlating current values on a stream with values that have previously occurred.

The result of applying a window over a stream is a time-stamped bag of triples over which conjunctions between triple patterns, and other "classical" operators can be evaluated. Windows can be converted back into a stream of triples by applying one of the window-to-stream operators in the `SELECT` clause: `ISTREAM` for returning all newly inserted answers since the last window, `DSTREAM` for returning all deleted answers since the last window, and `RSTREAM` for returning all answers in the window.

Table 3 shows a complete SPARQL$_{Stream}$ query which, every minute, returns the average of the last 10 minutes of wind speed measurements for each sensor, if it is higher than the average speed from 2 to 3 hours ago.

A.2 SPARQL$_{Stream}$ Semantics

The SPARQL extensions presented here are based on the formalisation of (Pérez et al.,). An RDF stream S is defined as a sequence of pairs (T,τ) where T is a triple $\langle s, p, o\rangle$ and τ is a timestamp in the infinite set of timestamps $\mathbb{T}$. More formally,

$$S = \{(\langle s, p, o\rangle, \tau) \mid \langle s, p, o\rangle \in ((I \cup B) \times I \times (I \cup B \cup L)), \tau \in \mathbb{T}\},$$

where I, B and L are sets of IRIs, blank nodes and literals. Each of these pairs can be called a "tagged triple".

A stream of windows is defined as a sequence of pairs (ω,τ) where ω is a set of triples, each of the form $\langle s, p, o\rangle$, and τ is a timestamp in the infinite set of timestamps $\mathbb{T}$, and represents when the window was evaluated. More formally, the triples that are contained in a time-based window evaluated at time τ □ $\mathbb{T}$, denoted ω^{τ}, are defined as

$$\omega^{\tau}_{t_s,t_e,\delta}(S)=\{\langle s,p,o\rangle \mid (\langle s,p,o\rangle,\tau_i) \in S,\ t_s \leq \tau_i \leq t_e\}$$

where t_s, t_e define the start and end of the window time range respectively, and may be defined relative to the evaluation time τ. Note that the rate at which windows get evaluated is controlled by the `SLIDE` defined in the query, which is denoted by δ.

Table 3: An example SPARQL$_{Stream}$ query which every minute computes the average wind speed measurement for each sensor over the last 10 minutes if it is higher than the average of the last 2 to 3 hours.

```
PREFIX fire: <http://www.semsorgrid4env.eu#>
PREFIX rdf : <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT RSTREAM ?WindSpeedAvg
FROM STREAM <www.semsorgrid4env.eu/SensorReadings.srdf>
    [FROM NOW - 10 MINUTES TO NOW SLIDE 1 MINUTE]
FROM STREAM <www.semsorgrid4env.eu/SensorArchiveReadings.srdf>
    [FROM NOW - 3 HOURS TO NOW -2 HOURS SLIDE 1 MINUTE]
WHERE {
 {
  SELECT AVG(?speed) AS ?WindSpeedAvg
  WHERE
  {
   GRAPH <www.semsorgrid4env.eu/SensorReadings.srdf> {
    ?WindSpeed a fire:WindSpeedMeasurement;
    fire:hasSpeed ?speed; }
  } GROUP BY ?WindSpeed
 }
 {
  SELECT AVG(?archivedSpeed) AS ?WindSpeedHistoryAvg
  WHERE
  {
   GRAPH <www.semsorgrid4env.eu/SensorArchiveReadings.srdf> {
    ?ArchWindSpeed a fire:WindSpeedMeasurement;
    fire:hasSpeed ?archivedSpeed;   }
  } GROUP BY ?ArchWindSpeed
 }
 FILTER (?WindSpeedAvg > ?WindSpeedHistoryAvg)
}
```

The three window-to-stream operators are defined as

$$\texttt{RStream}((\omega^{\tau},\tau)) = \{(\langle s,p,o\rangle,\tau) \mid \langle s,p,o\rangle \in \omega^{\tau}\}$$

$$\texttt{IStream}((\omega^{\tau}, \tau), (\omega^{\tau-\delta}, \tau-\delta)) = \{(\langle s, p, o\rangle, \tau) \mid \langle s, p, o\rangle \in \omega^{\tau}, \langle s, p, o\rangle \notin \omega^{\tau-\delta}\}$$

$$\texttt{DStream}((\omega^{\tau}, \tau), (\omega^{\tau-\delta}, \tau-\delta)) = \{(\langle s, p, o\rangle, \tau) \mid \langle s, p, o\rangle \notin \omega^{\tau}, \langle s, p, o\rangle \in \omega^{\tau-\delta}\}$$

In the above definition, δ is the time interval between window evaluations. Note that `RStream` does not depend on the previous window evaluation, whereas both `IStream` and `DStream` depend on the contents of the previous window.