Die approbierte Originalversion dieser Dissertation ist in der Hauptbibliothek der Technischen Universität Wien aufgestellt und zugänglich.
http://www.ub.tuwien.ac.at

The approved original version of this thesis is available at the main library of the Vienna University of Technology.
http://www.ub.tuwien.ac.at/eng

FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Semantic Stream Processing of Environmental Data

DISSERTATION

zur Erlangung des akademischen Grades

Doktor der Technischen Wissenschaften

eingereicht von

Dipl.-Ing. (FH) Peter Wetz
Matrikelnummer 0932123

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung:
O.Univ.Prof. Dipl.-Ing. Dr. techn. A Min Tjoa
Mag.rer.soc.oec. Elmar Kiesling, PhD

Diese Dissertation haben begutachtet:

O.Univ.Prof. Dipl.-Ing. Dr. techn. A Min Tjoa	O.Univ.Prof. Dipl.-Ing. Dr. techn. Andrew Frank

Wien, 16. Juni 2016

Peter Wetz

FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Semantic Stream Processing of Environmental Data

DISSERTATION

submitted in partial fulfillment of the requirements for the degree of

Doktor der Technischen Wissenschaften

by

Dipl.-Ing. (FH) Peter Wetz
Registration Number 0932123

to the Faculty of Informatics

at TU Wien

Advisor:
O.Univ.Prof. Dipl.-Ing. Dr. techn. A Min Tjoa
Mag.rer.soc.oec. Elmar Kiesling, PhD

The dissertation has been reviewed by:

O.Univ.Prof. Dipl.-Ing. Dr. techn. A Min Tjoa	O.Univ.Prof. Dipl.-Ing. Dr. techn. Andrew Frank

Vienna, 16th June, 2016

Peter Wetz

Erklärung zur Verfassung der Arbeit

Dipl.-Ing. (FH) Peter Wetz
Schumanngasse 3 / 7
1180 Wien
AUSTRIA

Hiermit erkläre ich, dass ich diese Arbeit selbstständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 16. Juni 2016

Peter Wetz

Acknowledgements

First and foremost I would like to thank my supervisor A Min Tjoa. Without his insights, comments, and suggestions this research would not have come to fruition in this way. His decades of experience and unique way of networked thinking opened up new perspectives and changed the way I approached new problems. The way he brings people together makes research more fun and having him supervise my work was truly inspiring. I would also like to thank Andrew Frank for revising this thesis and providing valuable feedback.

I want to thank Amin Anjomshoaa who warmly welcomed me at TU Wien, sparked first research ideas, and guided me in finding the right direction. His constant and never-ending friendliness and helpfulness will be something I will have fond memories of. I would also like to show my thanks to Elmar Kiesling, who became involved at a later stage of my studies. It was a pleasure to work with him and his always honest and detailed feedback significantly improved this endeavor. Likewise I want to thank my office colleagues from Graz (Hermann, Alfred, Vicky, Patrick, Jörg, Julia, Petra, Georg, and Angela) and Vienna (Lam, Dat, Peb, Bernhard, Lisa, and Andi). Without you my daily office routine in the past years would have been much less fun and exciting.

I would also like to thank my parents, who supported me during my studies at all times. They were always helpful, supportive, and respectful of what I was doing, which is something I am both happy and thankful for. My sincere thanks also go out to my girlfriend Lisa, who sometimes had to endure my moods due to bad days I had. I am officially sorry. Getting her feedback and having discussions with her was always fruitful and opened up new ways of thinking aside from my narrow and limited computer science perspective.

Kurzfassung

Da seit 2008 mehr als die Hälfte der Weltbevölkerung in urbanen Gegenden lebt, wird zu einem großen Teil in den Städten entschieden werden, ob wir die globalen Umweltprobleme lösen. Gleichzeitig wurde in der Wissenschaft in den letzten Jahren die Anwendung von Methoden der Informatik, um Umweltprobleme zu lösen, mehr und mehr erfolgversprechend. In dieser Arbeit stellen wir ein Verfahren vor, das das Ziel verfolgt, es BürgerInnen zu ermöglichen, gut informierte Entscheidungen auf Basis von Echtzeit-Umweltdaten zu treffen, um so zur Problemlösung beitragen zu können. Damit dieses Ziel erreicht werden kann, müssen Herausforderungen wie (i) die Integration von heterogenen Umweltdaten, (ii) die Identifikation adäquater Datenstrom-Management Systeme und (iii) das Design eines Systems zur effizienten Nutzung von Umwelt-Datenströmen durch städtische Interessensgruppen, gemeistert werden. Wir schlagen eine ontologie-basierte Methode vor, damit die stark heterogenen Umweltdaten integriert werden können. Zu diesem Zwecke führen wir ein neuartiges kontrolliertes Vokabular ein, das zwei de-facto Standards, nämlich die Semantic Sensor Network Ontology und das RDF Data Cube Vocabulary, kombiniert und erweitert. Um Datenstrom-Management Systeme zu evaluieren, erstellen wir ein Rahmenwerk namens YABench, das es erlaubt, verschiedene RDF Stream Processing (RSP) Systeme auf Basis unterschiedlicher Simulationsszenarien zu vergleichen. Dabei verwenden wir Parameter und Messzahlen, die vor allem im Umweltbereich wichtig sind, wie hochfrequente Messdaten, Korrektheit der Resultate (Precision und Recall) sowie Leistungsfähigkeit der Systeme (Speicherverbrauch und CPU-Belastung). YABench unterstützt somit die Entscheidung, geeignete RSP Systeme für unterschiedliche Umweltdaten-Szenarien zu ermitteln. Nachdem durch YABench C-SPARQL als passendes System identifiziert wird, stellen wir darauf aufbauend das Konzept “Linked Streaming Widgets” vor. Linked Streaming Widgets sind semantische Bausteine, die Streaming-Daten verarbeiten und von Anwendern zu Applikationen, sogenannten Mashups, zusammengesetzt werden können. Anwender können so Umwelt-Streaming-Daten integrieren und mit anderen Datenquellen verknüpfen, um letztendlich wohlinformierte Entscheidungen zu treffen. Wir setzen dieses Konzept als Erweiterung einer Mashup-Platform um. Des Weiteren präsentieren und diskutieren wir zwei Anwendungsfälle auf Basis von Citybike-Daten und Luftgütedaten und zeigen so die Durchführbarkeit des Konzepts. Eine abschließend durchgeführte Evaluierung demonstriert weiters die Praktikabilität der technischen Implementierung von Linked Streaming Widgets.

Abstract

Whether we cope successfully or fail to deal with the world's environmental challenges will be determined in cities where, since 2008, more than half of the global population resides. Recently, also the application of computer science methods to solve environmental issues is increasingly promising. In this thesis we present an approach to enable citizens to make well-informed real time decisions based on environmental data. To this end, we leverage semantic web technologies as a practical means to overcome the obstacles of (i) environmental data integration, (ii) identifying data stream management engines to process real time environmental data, and (iii) enabling efficient use of environmental data streams for city stakeholders. We develop an ontology-based approach to integrate highly heterogeneous and dynamic environmental data sources. We present a novel vocabulary that combines and extends two de-facto standard vocabularies, that is, the Semantic Sensor Network Ontology and the RDF Data Cube Vocabulary. Further, we create a framework to evaluate suitable RDF Stream Processing (RSP) engines based on the special requirements of the environmental data domain, such as processing of high-frequency data, providing correct results, and scalability. This framework called YABench facilitates the identification of appropriate RSP engines under varying circumstances for scenarios in the environmental domain. After we identify C-SPARQL as a suitable RSP engine, we propose "Linked Streaming Widgets". Linked Streaming Widgets represent lightweight semantic modules based on stream data, which can be combined to web applications by end users. By doing so, users can author their own mashups integrating environmental stream data sources, ultimately supporting well-informed decision making. We implement this concept as an extension of a mashup platform. To demonstrate its feasibility, we present and discuss two use cases based on citybike and air quality data, respectively, and perform performance evaluations indicating the practicability of Linked Streaming Widgets.

Contents

Part I

Analysis

CHAPTER 1

Introduction

This chapter presents the outline and motivation of this thesis. We show why the problem of integrating environmental data streams is relevant and state why the challenges related to this problem need to be tackled (Sections 1.1–1.2). These challenges include the integration of heterogeneous data sources, evaluation of stream processing engines, and the provision of data streams for users who lack programming expertise (Section 1.3). Further, we present the design science methodology used in the present work (Section 1.4) and the main contributions, that is, an ontology for environmental stream data, a benchmarking framework to evaluate stream engines, and a stream extension to a data integration platform (Section 1.5). Finally, we provide an overview of the structure of the present thesis (Section 1.6) and list peer reviewed articles which were published over the course of this research (Section 1.7).

1.1 Motivation

The environmental state of our planet has been rapidly deteriorating due to human activity. Despite technological achievements to minimize human impact on environmental processes, the combination of population growth and increasing consumption undermine the gains that have been made [Har98]. An annual report of the United Nations Environment Programme (UNEP), called Global Environment Outlook (GEO), points out that the "World Remains on Unsustainable Track Despite Hundreds of Internationally Agreed Goals and Objectives". Specifically, urban areas will have a large impact on the future development of threatened earth systems such as the atmosphere, biosphere, and hydrosphere [UNE12b, UNE12a]. As a consequence, we face a critical number of challenges which need to be addressed in order to improve the environmental state of the world. Computer science, as a means to deal with big volumes of complex data, are promising to address these challenge, such as global warming, pollution, and urbanization. However, the idea to deal with environmental issues with technological

means demands an increase for research and advances in the novel field of Environmental Informatics [HPRR95]. Researchers have already shown that methods and technologies derived from computer science have great potential [HC03].

In 2010, the global population living in urban areas surpassed 50% for the first time in history [Umw11]. The proportion of people living in cities is even higher in developed regions [UN12]. Therefore, it will be essential to our environment to find solutions in cities and view them as enablers for environmental change. Most harmful developments in urban areas are caused by people's behavior and affect, among others, air and water quality, noise pollution, climate and waste issues. The considerable impact of urbanization on global environmental issues, therefore, cannot be neglected. However, this can be seen as an opportunity to face these issues with techniques and technologies based on Information Technology (IT) methods enabled by and made available in cities. IT can act as a mediator to help tapping the full potential of urbanization.

This work contributes towards realizing the vision of a Smart City, a term which emerged in both the academic literature and the public discourse in the last years. One aspect of this vision is the availability and quality of Information and Communication Technologies (ICT) enabling, for instance, innovative traffic management, or providing environmental monitoring [CDBN11, HVMn+11]. One major barrier currently is that most urban developments are centered around vertical, i.e., domain- and industry-specific, solutions. These solutions foster data silos, a term derived from "information silos" [oFO88], and market islands. The goal is to provide horizontal services that make data available in a unified manner at urban-scale.

Another motivation of this work is the fact that urban areas continuously generate data due to their ubiquitous presence of sensors. Data is often considered as the new oil [RZB+12, Ken11, Pal06]. Availability and the ability to make use of data is a major advantage compared to rural areas, where a lack of data inhibits the creation of novel data services. This enables research towards the exploitation of the data generated by such devices which ultimately leads to innovative citizen services [RZB+12]. In such a city people are empowered to monitor their environment, retrieve real time data (traffic, pollution, public transport, etc.), and provide feedback to city managers leading to improved short-term decisions based on real-world conditions. Sensors can be any type of device measuring environmental values. This diversity consequently allows to tackle issues of energy, climate, mobility, and architecture. For instance, people who record energy consumption in homes gain detailed insight into their consumption behavior which allows them to better manage electricity costs. Similarly, traffic sensors record traffic flows and allow the development of services which improve routing algorithms and ultimately lead to less congestion especially in cities. Sensor data is often continuously generated and available immediately, however, means to exploit it are sparse. Availability of raw data is, therefore, only a first step that has to be followed by careful processing and addition of contextual information to enable the implementation of new applications.

By following the Smart City vision, new developments can contribute to a more comprehensive understanding of the world around us. The focus of this work is to raise

awareness for environmental challenges. Services built upon ICT and ubiquitous sensing will facilitate city stakeholders, i.e., city managers, policy makers, citizens, with a means to make well-informed decisions, taking into account environmental factors based on the current monitored state of the city. The underlying idea follows a quote of Thomas Goetz, who wrote in his Wired article [Goe11]: *'Provide people with information about their actions in real time, then give them a chance to change those actions, pushing them toward better behaviors.'*

Finally, the European Union (EU) Directive from 2003 [Uni03] is a key motivation. This directive regulates the right of citizens of the EU to obtain access to environmental information. Every citizen should have means to easily and continuously access such information as much as possible in a systematic way, particularly through the use of ICT [Uni03]. However, from a technical perspective, there are critical barriers to access environmental information in an easy way, thwarting the initial aims of the EU.

Current deficits and challenges include the following: (i) environmental information is distributed via multiple agencies exacerbating access to the data, (ii) the data is stored heterogeneously, i.e, without a standardized presentation, (iii) often, citizens do not know which agency they can request certain data from, (iv) the currently implemented process does not account for the increasing importance of real time data, and (v) utilization of the "context" of the data lacks, that is, providing additional information related to the data, which can be used for more sophisticated services.

There are several challenges in providing large-scale environmental information to the public. The proposed research addresses them to enable comprehensible, comparable and unified access to up-to-date environmental data.

By combining the impact of urbanization on our environment and the right of environmental information on the one hand, and the enabling technologies in the context of smart cities, on the other hand, this thesis proposes an approach that allows to process pervasive environmental data in real time. An abundance of data generated in a city are available, but they are not used to their full potential. Therefore, we propose a unified system offering intelligent exploration of environmental data via sensors. By doing this, we address the presented challenges. Stakeholders are then able to use data sources to observe, aggregate, and visualize data as it flows in to change their behavior. This will help people to make use of previously meaningless, but still pervasively available, environmental data.

This work aims at allowing people in an urban environment to make decisions concerning environmental aspects of their surroundings based on timely data. Semantic processing of the data enables to (i) create a homogeneous data repository and (ii) construct queries mashing-up different data sources. This work will extend the value of urban sensor data, which is currently limited due to its confinement in vertical applications.

1.2 Aim of the work

The aim of the present work is to create horizontal services, i.e., services that make data available in a unified manner, and applications that use federated and aggregated sensor data across domains enabled by semantic technologies. Such services can focus on the needs and interests of users, e.g., as data streams can be discovered and used based on contextual information extracted from the stream's semantics.

As a result the user may just discover data from his proximity, based on his interests (e.g., air quality observations), based on time constraints (e.g., last 30 minutes), or a combination of these (e.g., air quality sensor observations of the last 30 minutes as near as 100m to the user). Even discovery based on current values, aggregates (sum, median, mean, mode, min, max, etc.) or trends (increasing trend, decreasing trend, stagnating) poses an interesting opportunity for city stakeholders, e.g., one may be interested, if the average speed of cars at the road to his/her home in the last 30 minutes has fallen below a certain value.

Furthermore, since we will follow a graph-oriented approach which enables data to be interlinked, utilization of available, but currently isolated data will be extended into the following directions: (i) streams automatically will obtain a context-aware characteristic, for instance based on geographic location, sensor interest or thematic interest, which enhances stream discovery and stream querying, (ii) federated queries provide a means to combine different data streams into one query, and (iii) continuous queries can be enriched with background knowledge sourced from static knowledge repositories.

There is a large amount of environmental data which gets continuously generated and there are semantic web technologies, which can help to deal with this data effectively. Our investigation will aim to connect these aspects to better understand the dynamic nature of our environment.

More specifically, the concept of Linked Data Streams, which is a recent and not yet well explored research area, has not been applied to environmental sensor data in an urban context. Therefore, this thesis contributes to the current knowledge in three ways: (i) Proposing an approach to abstract and map observation data to a formalized vocabulary, (ii) provide a means to identify suitable data stream processing engines, and (iii) the development of a prototypical system for semantic stream processing of urban data sources. The ultimate goal is to democratize the usage of stream processing through analysis with a simple user interface.

1.3 Problem Statement

The thesis tackles the challenge to make real time environmental data of a city usable for its stakeholders. Therefore, the overall problem to be addressed is:

- Can real time environmental data be provided to create actionable knowledge for urban stakeholders?

Due to the heterogeneity of environmental data, an initial step is to integrate it. Heterogeneity can be technical, structural, and semantic. Overcoming the former two is possible by applying data transformations and schema mappings. However, the issue of semantic heterogeneity is needs to be addressed in order to solve the stated problem. Resolving semantic heterogeneity would pave the way for developing a common machine-understandable method to process previously semantically disparate data sources. Addressing this is non-trivial and requires the definition of a common formal understanding of the domain at hand.

Generally, data integration has been a well-studied problem over the last decades [Hul97, Ull97, HRO06]. Thus far, however, no satisfying solution in the context of real time and semantically heterogeneous data could be found. We define the city of Vienna as the area this research will be applied on, because it has an active Open (Government) Data community, performs excellently in recent Smart City Rankings [Coh14], and runs a courageous Open Data Platform[1].

1.3.1 Integrating Heterogeneous Data Sources

The first challenge, which we deduce from the overarching problem statement is the integration of heterogeneous data. By heterogeneous data we mean data derived from different sensors, such as continuous environmental data. We therefore formulate:

RQ1 *What data integration methods can be used to model environmental real time data to overcome heterogeneity and to allow reusability and explorability?*

We aim to provide a system, which allows to grasp the current state of a city. Hence, we need to deal with a vast amount of different data sources. Initial investigation revealed that there are numerous data sources of interest available, but they are provided in a highly heterogeneous manner. Mainly, the sources differ in the following characteristics:

- Data type
  - RDF
  - JSON
  - XML
  - CSV
  - PDF
  - HTML
- Data access

[1] `https://open.wien.at/site/open-data/` (accessed 16 June 2016)

  - API
  - File download
  - Manual crawling
- Update frequency
  - static, i.e, rarely updated
  - hourly/daily/weekly/... updated
  - stream data, i.e., real time updated

As a result of these variations, highly diverse data is available. Integration of these data sources provides the basis for further utilization of the data streams.

Ontologies have been a well-known tool to tackle data heterogeneity for years [NM01, Hul97, Gru93]. They represent formal vocabularies of a specific domain. Applications can reuse and process information from different sources, if they share a common understanding of the structure of information. This approach is called *Knowledge Sharing*. We follow this vision of Knowledge Sharing because environmental data coming from different sources can be collated by a common domain ontology. Hence, this will form the basis for further processing of the data. Comprehensive and standardized semantic models are required to power semantic search (e.g., discovery of sensors serving context-sensitive data) and knowledge extraction (e.g., detect event patterns such as decreasing air quality) from sensor generated data. Essentially, we need to answer questions about the required granularity and complexity of the data that is modeled in the ontology. Furthermore, we need to decide if a lightweight model fits, or if heavyweight and formal semantics need to be implemented. There are strengths and weaknesses to each approach [UG04] which we need to evaluate in detail.

Different approaches to model continuous sensor data have already been proposed, e.g., the Semantic Sensor Network Ontology (SSNO) [CBB+12]. Yet, due to the generic descriptions of observation and measurement data offered by the ontology, it cannot be used to annotate data with domain specific knowledge. What we want to achieve in this work is to enrich the data with domain specific context and lift it to a higher semantic level. Another limitation is that it is mainly designed to perform reasoning for sensor-related descriptions rather than its observations. Furthermore, its applicability regarding challenges for semantic web sensors still has to be investigated [CG10].

The recently proposed RDF Data Cube Vocabulary is an approach to model measurement and observation data [CR14]. However, the extent of *Knowledge Reuse* is still an open issue, i.e., which vocabularies can be reused and which parts need to be modeled from scratch. A major advantage of *Knowledge Reuse* is that existing concepts can be reused by interlinking with existing ontologies. Another advantage of an ontology-based approach is *Reasoning*, i.e., inferring novel implicit facts based on explicit knowledge. Lastly, this method enables aggregations of sensor observations across domains, allowing for intelligent data combination and knowledge extraction.

Finally, semantic enrichment of environmental smart city data streams accounts for heterogeneity of incoming raw data observations and will, thus, enable interoperability and further exploitation of available data streams. Furthermore, we aim to advance the state of the art by providing high quality Linked Data (see [Ber06]) in the domain of Linked Stream Data (LSD), hence, moving one step towards the Web of Data vision. The process of lifting diverse data sources on a common higher level, i.e., data harmonization, has the potential to improve both data reusability and data value.

1.3.2 Evaluation of Stream Processing Engines

The second subproblem is to evaluate stream processing systems which can subsequently be used to provide the data to the user. Common solutions in the domain of the semantic web aim to offer static data or data which does not change frequently. In contrast, we have to deal with permanently updated and continuous streams of data. Therefore, this problem can be formulated as follows:

RQ2 *How can we evaluate systems for semantic processing of real time environmental data streams?*

Different approaches in evaluating semantic stream processing systems have already been proposed, but their scope is limited and they do not cover the whole process from scenario description to results visualization. Consequently, designing a method to evaluate real time stream processing systems, namely C-SPARQL and CQELS, is the resulting challenge.

This challenge can be investigated from two perspectives. First, in the literature Data Stream Systems are proposed [BBD+02]. Although they partially introduce continuous queries, scalable processing, and handling of real time data, they rely on a relational database paradigm that lacks semantic query languages and ontological foundations. Hence, they cannot be utilized to leverage integrated data streams from different sources. Second, from the semantic perspective the typical approach to face provision and access to data is the Simple Protocol and RDF Query Language (SPARQL) Protocol and RDF Query Language (SPARQL and SPARQL 1.1, respectively), which is a World Wide Web Consortium (W3C) Recommendation to query Resource Description Framework (RDF) graph data [HS13, PS08]. This provides the ability to utilize a query language and the power of ontologies. When applied to the proposed work, SPARQL has one substantial limitation in that it is tailored towards dealing with static knowledge bases and not geared towards knowledge evolution based on continuous querying and changing data. SPARQL does not allow to introduce continuous semantics in a real time environment.

Efficient means to process semantic data streams are required in order to combine the advantages of both presented perspectives. Recently, efforts toward this issue have been proposed in research: C-SPARQL [BBC+10c], CQELS [LDPH11], SPARQL-stream [CCG10] and Morph-Streams Processor [CCJA12], EP-SPARQL [AFRS11] plus

ETALIS [AFR+11], and INSTANS [RNT12]. Moreover, at the W3C, an RDF Stream Processing Community Group[2] has been formed to deal with the definition of a common model for RDF streams to facilitate provision, transmission, and continuous queries.

Evaluating semantic stream processing engines is essential for the implementation of a system which should be capable of integrating heterogeneous data streams. Scharrenbach et al. [SUM+13] present three key performance indicators that help in defining stress tests for the evaluation of stream processing frameworks. The key performance indicators are (i) *response time over all queries*, (ii) *maximum input throughput*, and (iii) *minimum time to accuracy and the minimum time to completion for all queries* (includes precision, recall, and error rate).

By leveraging RDF Stream Processing, our system will be able to provide efficient, federated, and time-dependent queries over arbitrary data sources, while it makes use of the added semantics. Furthermore, this will empower users to combine static or historical background knowledge with real time data sources, entailing novel use cases and applications while using the data. To this end, a key goal is to identify which stream processing proposal is best-suited for our approach.

1.3.3 Leveraging Data Streams for City Stakeholders

The final challenge derived from the initial problem statement is how to let users make use of data streams. After having identified a means to process semantic data streams, interested stakeholders should be able use such streams to gain new knowledge. We aim to capture the changes of environmental city data and enable corresponding services to the users of the final system. These requirements can be collected under the term *Stream Processing* or *Stream Reasoning*, i.e., means that allow to infer new knowledge based on continuous data streams [VCHF09].

When living in a large city with much traffic, it would, for instance, be helpful to provide services which help citizens to circumvent traffic jams, predict their impact, and even recommend alternative routes. This vision goes beyond typical data mining techniques which enable to detect traffic jams. Providing more sophisticated analysis incorporating pollution data and putting real time data into context to citizens and city managers are more challenging and require more complex approaches than the state of the art. Or if newcomers to a city are looking for an apartment, which should be both quiet and favorably located. Based on environmental urban data streams a service could provide information on which locations are noisy based on real time data. Even recombining different data sources, e.g., air quality, weather and traffic data, to uncover new facts would enable innovative analysis. Based on weather phenomena (wind speed, rainfall, snowfall, etc.), detection of specific events (whirlwind) can be enabled. Currently, there is no system that allows for such complex queries, let alone enables ordinary users to do so in a real time fashion exploiting the accessibility and flexibility of web technologies [MUHB14]. As a result, we define the following research question:

[2] `http://www.w3.org/community/rsp/` (accessed 16 June 2016)

RQ3 *How can non-expert users, i.e, urban decision makers, be enabled to explore environmental stream data?*

To address this question of leveraging available data streams, reasoning approaches will be utilized. More precisely, stream data features of SPARQL extension proposals will be investigated and applied, i.e., windowing functions and federation of static data with dynamic streams, to achieve time-varying inferences. To extend available approaches, and to contribute to the current state of the art, this work will combine stream reasoning techniques with a widget-based approach. Similarly to Web Services [ACKM04], widgets are small web applications that fulfill small well-defined tasks. Each data stream is represented as a widget and implements an innovative interface that allows intelligent coupling of such widgets combined with processing functions, again modeled as widgets. Hence, users will have the power to efficiently exploit available data streams in a real time manner.

These processing widgets have encoded queries based on stream specific criteria, e.g., time windows or aggregates (sum, count, average, etc.), and therefore return RDF triples that answer this query. So called presentation widgets, will also be needed to visualize the output via maps, bar charts, line charts, pie charts, and histograms. This step covers three aspects of leveraging data streams: (i) *analyzing* via continuous stream queries, (ii) *publishing* via returning RDF graphs, and (iii) *visualizing* via corresponding presentation interfaces.

1.4 Methodology

We adopt the design science research methodology. Design science is important in disciplines where the creation of artifacts is essential. This is often the case in an application-oriented IT contexts [PTRC07]. Furthermore, in design science knowledge and understanding of a problem domain and its solution are achieved in the building and application of the designed artifact [vAMPR04]. The methodology is a four phased process consisting of (1) Theory (2) Design (3) Evaluate and (4) Justify, combining theoretical and practical aspects. The result will be an IT artifact solving a class of problems. Figure 1.1 provides an overview of how design science will be applied in this thesis.

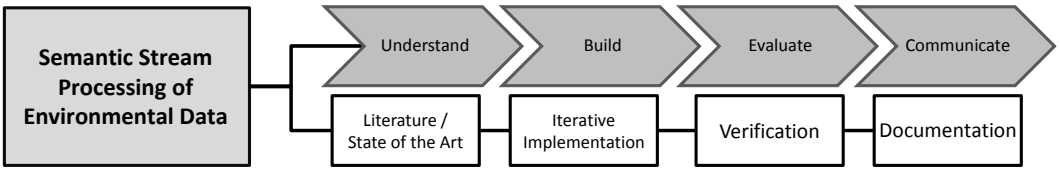


Figure 1.1: Design Science Approach (adapted from [PTRC07])

In the first phase we will review the available literature of the related research fields. This will map previous research efforts from two perspectives. First, we will review literature on semantic stream processing. Second, the review will cover activities from the Environmental Information Systems field related to modeling environmental data. This will help to get an understanding of the related research fields and to identify gaps which have not been addressed.

Based on this, the build phase consists of an iterative implementation activity, dealing with the presented problem statements. We will identify suitable data sources, evaluate available systems able to process environmental streams, and implement an approach to utilize them. Ontology engineering methods will be applied and stream processing systems have to be evaluated to guarantee a scalable and useful solution to the central research problem. This phase will result in a prototypical implementation of a system aimed to address and solve current challenges in environmental data stream processing.

In the evaluation phase characteristics of the prototype will be analyzed in terms of performance and scalability. This will allow us to draw conclusions regarding the feasibility of our approach and will help to compare it to similar works. Finally, results will be disseminated through publication of papers at workshops, conferences, journals, and, ultimately, a PhD thesis.

To sum up, each step of the four phases will contribute to the building of artifacts, which help us to understand the problem domain and to solve the described challenges. Especially the *Build* and *Evaluate* phases are crucial for the design and implementation of our solution approaches. This helps us to iteratively advance towards finding detailed answers to our research questions.

1.5 Contributions

In the following we describe the central contributions of this thesis. First, a novel semantic data model for environmental stream data is designed to be able to timely integrate available raw data streams. To make use of the stream data model, we need to evaluate existing semantic stream processing systems with respect to their suitability to process environmental data streams. In this step we take into account the special requirements which are imposed by environmental data, such as the high frequency of data arrival, necessity of correct results, and the ability to provide results as fast as possible. To this end, we develop a framework which is enables to generate and conduct stream processing experiments and visualize the results. Second, based on our findings from the data model and experimentation, we present an approach which hides the complexity of the data model and stream processing and allows end users that lack programming knowledge to make use of environmental data streams.

Environmental stream data model. By both defining the requirements to environmental stream data and by knowing the principle of reuse in the semantic web, we

identify a need to bridge two vocabularies of this field having a semantic overlap. Namely these are the Semantic Sensor Network (SSN) ontology and the RDF Data Cube Vocabulary. We analyze issues which arise when using these ontologies with respect to stream data and create a novel vocabulary via extending and reusing both. This vocabulary provides a lightweight method to model streaming environmental data sources. Further, it enables to semantically describe such sources and facilitates processing, reuse, and discoverability. Moreover, the semantic enrichment of raw data streams fosters integration of disparate sources and integration of slowly changing static knowledge sources.

Stream data scenario generation. When stream processing engines are evaluated, the input data and the complexity of its model strongly influence the results which will be returned. Feeding data which is not suitable to gain relevant insights into an engine's performance with respect to the applied domain can render the measurements useless. To this end, we provide a means to flexibly generate stream data sources in order to emulate different scenarios of interest in a streaming setting. This approach allows to define input data of varying size, speed, complexity, and arrival time distribution. In future, researchers will be able to use this approach to define scenarios which fit the domain they investigate. This makes their results more reliable and useful.

Stream processing evaluation framework. We provide a comprehensive framework to conduct evaluations of semantic stream processing systems based on streaming scenarios which can be defined as desired. The presented benchmarking framework allows us to interpret a system's behavior and to quantify and characterize its results along various dimensions. These dimensions include performance characteristics (memory consumption, CPU load, etc.) as well as correctness measurements with regard to result delivery (precision and recall). The framework covers all stages of benchmarking starting from data generation, performing an experiment, and presenting its results. This is the first benchmark proposal for semantic stream processors comprehensively complementing an engine's characteristics into a consolidated view. Ultimately this allows to gain detailed insights into the performance and behavior of the tested engines. The use of this framework enables us to identify the most suitable system for our domain of environmental stream data processing.

Streaming mashups architecture. An innovative aspect of our work is the implementation of an architecture which facilitates end users to compose mashups based on stream data. A key challenge is to abstract away the complexity which is introduced by stream processing. To overcome this barrier we implement a method which transforms input data streams to RDF data which, in turn, is consumed by streaming widgets. These widgets provide a simple user interface and are semantically annotated to support users in semi-automatically building meaningful mashups. Hence, we show how the complexity of the data model can be hidden,

while still providing the users with a capability to compose mashups by integrating and enriching available data. Furthermore, this enables end users to integrate stream data with static data sources fostering novel insights based on real time environmental data streams.

1.6 Structure

Chapter 2 introduces an ontology for the modeling of environmental stream data. We elicit the requirements, discuss design considerations, and present the final vocabulary which is subsequently used as a data model to represent environmental data streams.

Chapter 3 describes the developed framework to benchmark semantic stream processing engines. The framework is used to investigate which of the proposed engines is the most feasible for the implementation of linked streaming mashups.

Chapter 4 introduces linked streaming widgets as a means to process stream data. We present our extensions to the Linked Widgets Platform and showcase how streaming widgets can be used to provide real time data to end users. We also provide an evaluation on the performance characteristics of our approach.

Chapters 5–7 summarize our work and provide answers to the initially stated research questions. Furthermore, directions for future work which could not be covered in this work are discussed.

Sections on related work which reflect on the state of the art and highlight our contributions against related approaches are embedded as subsections in Chapters 2–4.

1.7 Publications

This thesis includes work and results from the following peer-reviewed publications ([WAT13, WTD+13, WTD+14, WTD+16, KW15, KWA+16]):

- Wetz, P., Anjomshoaa, A., Tjoa, A M. (2013), A Survey on Environmental Open Data in Austria. In Proceedings of the 2013 IEEE International Conference on Systems, Man, and Cybernetics (SMC2013), IEEE, pp. 4566 – 4570. http://dx.doi.org/10.1109/SMC.2013.777.
- Wetz, P., Trinh, T. D., Do, B. L., Anjomshoaa, A., Tjoa, A M. (2013), Austrian Environmental Data Consumption — A Mashup-based Approach. In Proceedings of the 1st International Workshop on Semantic Machine Learning and Linked Open Data (SML2OD) for Agricultural and Environmental Informatics co-located with the 12th International Semantic Web Conference (ISWC 2013).

- Wetz, P., Trinh, T. D., Do, B. L., Anjomshoaa, A., Kiesling, E., Tjoa, A M. (2014), Towards an Environmental Information System for Semantic Stream Data. In Proceedings of the 28th Conference on Environmental Informatics – Informatics for Environmental Protection, Sustainable Development and Risk Management (EnviroInfo 2014), BIS-Verlag, Oldenburg, pp. 637 – 644. http://dx.doi.org/10.1037/rmh0000008. **Best Paper Award**

- Wetz, P., Tuan, D. T., Lam, D. B., Anjomshoaa, A., Kiesling, E., Tjoa, A M. (2015), Towards an Environmental Decision-Making System: A Vocabulary to Enrich Stream Data. In Advances and New Trends in Environmental and Energy Informatics, Springer, ISSN: 2196-8705, pp. 317-335. http://dx.doi.org/10.1007/978-3-319-23455-7_17.

- Kolchin, M., Wetz, P. (2015), Demo: YABench-Yet Another RDF Stream Processing Benchmark. In RDF Stream Processing Workshop co-located with the 11th Extended Semantic Web Conference (ESWC 2014), Portoroz, Slovenia.

- Kolchin, M., Wetz, P., Kiesling, E., Tjoa, A M. (2016), YABench: A comprehensive framework for RDF stream processor correctness and performance assessment. In Web Engineering – 16th International Conference on Web Engineering, ICWE 2016, Lugano, Switzerland, June 6-9, 2016, Proceedings, Springer Berlin Heidelberg. *accepted*

Part II

Design

CHAPTER 2

Modeling Environmental Data Streams

In this chapter we present the design of an ontology to model environmental stream data. We present related work (Section 2.2), elicit the requirements, evaluate existing ontologies (i.e., the Semantic Sensor Network Ontology and the RDF Data Cube Vocabulary), and discuss design considerations (Section 2.3). Subsequently, we show how the developed ontology relates to the overall architecture developed in this thesis (Section 2.4).

2.1 Motivation

Methods based on IT can support city planners and assist in real time environmental decision-making, such as, for instance, analyzing traffic and air pollution data streams to dynamically optimize traffic routing. Such applications are rendered possible by the ubiquitous presence of data stream-generating sensors. Integrating this real time sensor data can lead to innovative citizen services and will ultimately help to trigger change in how we interact with the environment [RZB+12]. However, available techniques to exploit continuously generated environmental data streams are still limited and existing environmental information systems that monitor air pollution, water quality, or transport systems based on real time data operate only within the isolated scopes of their domain. Abundant raw data is only a first step; to extract relevant insights, it is necessary to enrich it with contextual information, integrate it with data from other streams, and carefully process and analyze it.

In this chapter, we present related work and state of the art in environmental data modeling to facilitate processing of heterogeneous environmental data streams. We then conceptualize and outline a vocabulary that captures concepts which are required to model environmental stream data. This vocabulary is used in an architecture of a web-based

platform for the semantic integration of heterogeneous environmental data sources in real time. This platform provides a unifying layer that integrates diverse environmental city data. In particular, we tackle three main challenges, i.e., to (i) integrate data originating from different sources and formats, (ii) facilitate semantic querying of the integrated stream data following linked data principles, and (iii) address information needs in the environmental domain in real time.

2.2 Related Work

Using ontologies to model data from different sources has been studied for years [Noy04, MIKS00, WVV+01, Gag07, BCVB01, HG01]. Ontologies allow to develop precise domain models for a given field of interest. Given an agreement on an ontological model, the semantics of data provided by data sources for integration can be made explicit. Hence, based on this shared understanding, challenges due to semantic heterogeneity are reduced.

In the particular field of this work, that is semantic modeling of environmental observations, applying ontologies to align data sources has been considered [PB02, MFC+07, MRS+07, MRM+10]. However, these works focus on a more coarse level of data integration, meaning they ease semantic integration on a data provider level (databases, files, repositories, etc.). Our data model operates on the granular level of environmental observations. This allows to improve the ways of interacting with the data in three ways: First, data access is facilitated, because applications can use the ontology as a single point of access. Second, querying can be performed by using the vocabulary provided by the ontology for unified query formulation. Third, given that ontologies can be designed in order to be easily extensible, new data sources can be added rapidly.

2.2.1 RDF Data Cube Vocabulary-based Data Modeling

The RDF Data Cube Vocabulary [CR14] has been used to publish semantically explicit integrated environmental data sets as linked data. For instance, [RFH10] provide vocabularies in the field of environmental observation to enable the long-term analysis of substance exposure of humans and the environment based on linked data repositories. They use a set of domain ontologies to encode information of relevant application fields, such as biodiversity or substances. A global ontology is connected to these domain ontologies through predefined properties. However, this approach is not generic, because it uses properties which are specifically defined for the publication of Environmental Specimen Bank (ESB) data.

Another example for the use of ontologies for environmental data integration is the UK government. They release data about the bathing water quality utilizing the RDF Data Cube Vocabulary. The bathing water measurements are also linked to spatial data of the UK Ordnance Survey and displayed on a map which is updated regularly[1]. This example

[1] `http://environment.data.gov.uk/bwq/profiles/` (accessed 16 June 2016)

demonstrates how environmental data can be enriched via interlinking with other data sets and how it can be visualized in an appealing way.

Tarasova et al. [TAM13, Tar13] reuse the RDF Data Cube vocabulary to model heterogeneous environmental data sources. The authors process static data sets of historic data and harmonize them by introducing a semantic metadata model that focuses on a generic and domain-independent solution. This approach eases retrieval and integration of measurements from different data collections at a single point, that is the presented ENVRI portal.

2.2.2 Semantic Sensor Network Ontology-based Data Modeling

The SSN ontology aims "to describe sensors and sensor networks for use in sensor network and sensor web applications" (see [LHT11]). This ontology relies on design patterns which draw from the functional approach presented by Kuhn [Kuh09, JC10]. Researchers use SSN to integrate (e.g., [FMMVA13, LK14] and publish (e.g., [MFM11, JBS+13] environmental sensor measurements as linked data.

In [BP10] the authors describe Sense2Web[2], a linked data platform to publish sensor data. This platform allows applications and users to request measurements and observation data and to query events based on semantic web principles. The authors state that the advantage of this approach lies in improved data access and querying capabilities. This allows to obtain information or integrate data from various sources. An example access and exploitation scenario for the constructed linked sensor data platform is also described using a mashup application.

In [LPQPH11] a Linked Sensor Middleware based on the SSN ontology is presented. The authors introduce LSD which are essentially sensor stream sources enriched with semantic descriptions. The developed Linked Stream Middleware platform is used to process LSD allowing for collecting, publishing, annotating, visualizing, and querying stream data.

Context-aware Sensor Search, Selection, and Ranking Model (CASSRAM) builds upon the SSN ontology and facilitates the discovery of sensors in the Internet of Things (IoT) domain through exploitation of context information [PZC+13]. The authors present a prototype tool which utilizes the model and shows how semantic and statistical reasoning can be combined to improve the data acquisition process. The tool is evaluated with respect to performance and response time revealing a trade-off between result accuracy and resource consumption. Machine learning techniques are proposed to learn optimal margin of error parameters in order to optimize the trade-off with respect to user requirements.

Further, the SSN ontology is used for modeling environmental sensor observations in

[2] `http://personal.ee.surrey.ac.uk/Personal/P.Barnaghi/Sense2Web.html` (accessed 16 June 2016)

several research projects, such as SemsorGrid4Env[3], Aemet Linked Data[4], Exalted[5], and Spitfire[6].

2.2.3 Research Gap

As shown in Table 2.1, all of the previous research focused on either the RDF Data Cube vocabulary or the SSN ontology. Moreover, we see that, with one exception, all works which use the RDF Data Cube Vocabulary process measurement data, whereas all approaches which use the SSN ontology use the data model to describe the metadata. This shows that there is a lack of a vocabulary that allows to use the RDF Data Cube Vocabulary and SSN on the same data structures. This would create compatibility at the data layer as well as a higher level of reuse. Such a vocabulary offers means to describe sensors and the data they observe, while it can be queried with query patterns which are typically used with the respective ontologies.

Moreover, following best practices of semantic web data publishing, vocabularies which have a semantic overlap, as is the case for Data Cube and the SSN ontology, should be aligned to enable reuse. Such an alignment can then, for instance, allow to query respective data sources based on both ontologies. In addition, if a data publisher wants to provide sensor data as RDF Data Cubes, it is essential to find a mapping between these two vocabularies. To achieve this alignment, a means to store observational data corresponding to both vocabularies need to be found.

Initial work on combining both ontologies has already been done. Lefort et al. [LHTW13, LBH+] aim to align them in order to convert a historic climate data set to RDF. They

Related Work	Ontologies		Data Layer	
	RDC	SSN	Measurements	Metadata
[RFH10]	✓	-	✓	-
[TAM13, Tar13]	✓	-	✓	-
[Env]	✓	-	✓	-
[BP10]	-	✓	✓	✓
[LPQPH11]	-	✓	-	✓
[PZC+13]	-	✓	-	✓

Table 2.1: Overview of literature in ontology-based environmental data modeling[7]

[3] http://www.semsorgrid4env.eu/ (accessed 16 June 2016)

[4] http://aemet.linkeddata.es/ (accessed 16 June 2016)

[5] http://www.ict-exalted.eu/ (accessed 16 June 2016)

[6] http://web.archive.org/web/20150227022003/http://www.spitfire-project.eu/ (accessed 16 June 2016)

[7] The *Ontologies* column denotes which ontologies are used by the respective approaches (*RDC* is the RDF Data Cube Vocabulary, *SSN* is the Semantic Sensor Network Ontology). The *Data Layer* column shows on which level the presented work operates (*Measurements* denotes use on the instance level, *Metadata* shows that the data model is used to describe the structure of the resources).

provide a web-interface to browse the data based on the encoded semantics. However, the vocabularies are coupled more loosely than is the case in our work. The authors use the SSN ontology to publish metadata and the RDF Data Cube vocabulary for observation data. Their work is based on the premise that the declarations of observed properties in the SSN ontology as classes are not directly compatible with their declarations in the Data Cube vocabulary as properties. In our work, we did not have to tackle this issue, because we model the data structure in a way that complies with both approaches (cf. Section 2.3.2.4).

Stocker et al. [SRK14] design an ontology that aligns and specializes the generic concepts of several upper ontologies, i.e., SSNO, DOLCE+DnS Ultralite (DUL), RDF Data Cube Vocabulary (QB), Situation Theory Ontology (STO), GeoSPARQL, Time Ontology in OWL, and PROV Ontology (PROV-O). However, they also model *ssn:Observation* as having a different semantics than *qb:Observation*. The former represents a sensor observation that relates to the sensor that made an observation (result of sensing); the latter represents a data set observation, for instance, a line of a textual observation file (result of computations). Since their work aims at modeling a multi-step process of environmental monitoring this separation is suitable in this particular case.

To conclude, modeling of environmental data has been a research topic for many years. More recently, the use of semantic web techniques in order to model and process environmental observations to facilitate integration of increasingly dispersed data sources has gained interest. Most prominently researchers and developers use the RDF Data Cube vocabulary and the SSN ontology to store such data. However, since there is a semantic overlap between these two vocabularies, there is a need to bridge similar concepts of both to maximize compatibility and possibilities of reuse. This will, for instance, enable the formulation of unified queries over environmental data sources based on both vocabularies. Moreover, if this semantic gap would be removed, data cubes can be automatically generated based on SSN linked data. We address this issue in the course of this thesis (cf. Section 2).

2.3 Ontology for Modeling Environmental Data Streams

The design of an ontology to model and annotate environmental data streams is a prerequisite for the overall architecture and a central contribution of this thesis. Ontologies form an integral part of the semantic web stack [BL03]. They are used to describe data and explicitly define their semantics. Due to the heterogeneity of environmental data, it is not feasible to design a single ontology that covers all environmental domains. Therefore, the linked data principles propose to use multiple coexistent vocabularies to describe data. Hence, our ontology represents an extensible vocabulary for sensor observations in the environmental domain, which can be complemented with external information.

The main goal of our environmental data model is (i) to provide a concise, but complete vocabulary to model, annotate and semantically enrich environmental data streams while (ii) reusing already existing ontologies wherever possible. Since the SSN ontology and

the Data Cube vocabulary have overlaps in their fields of application, i.e., capturing (sensor) observations, this is the first approach to combine and align both ontologies while preserving and respecting the definitions of the concepts. Our integrated ontology allows for queries that exploit the ideas, notions, and use cases of both source ontologies.

The following sections provide a detailed overview on the design process, requirements, reused ontologies, and key considerations. We did not use any of the well recognized ontology engineering approaches, such as Methontology [FLGPJ97], On-To-Knowledge [SSS04], and DILIGENT [PST04], because they require that an ontology is built from scratch. Instead, we use methodological guidelines of the *Building Ontology Networks by Reusing Ontological Resources* scenario [FLSFGP12] which was developed as part of the NeOn methodology [SFGPFL12]. The ontology discussed is actively maintained and available online[8]. In the present work we use the prefix *es* (Environmental Streams) to refer to terms of this ontology.

2.3.1 Requirements

Environmental data streams are heterogeneous and vary along several dimensions, e.g., data type, update frequency, and covered domain. To process them in a homogeneous manner, the definition of a unifying data model is therefore essential. We choose ontologies as a foundational data model to describe our domain of interest, i.e., environmental data. Ontologies are highly useful for data integration tasks and to enable unified data access [CX05].

Incoming data is converted to a data model that conforms to a described ontology and therefore enables access to the previously separated sources by means of a controlled vocabulary. The conversion process semantically enriches the data, because raw data gets annotated using well-defined domain concepts. In other words, meaning is added to the data which can later be exploited by applications and humans. Since the enriched concepts are both machine- and human-readable, they can be reused by semantic clients as well as by humans. Semantic enrichment is a fundamental requirement to realize consecutive steps of the presented architecture, i.e., data streaming and stream processing, in order to create a system which can cope with heterogeneous data streams. Other tasks to realize are contextualized sensor discovery, semantic search and query mechanisms, and knowledge extraction from sensor-generated data.

We design an ontology for the environmental domain based on the following set of requirements:

R1 – Data integration. The ontology should facilitate integration of different data sources that capture environmental observations. Therefore, it should capture knowledge about phenomena that appear in our environment. Moreover metadata about sensors (update frequency, location, etc.) and observations (unit of measure,

[8] `https://github.com/beta2k/environmental-stream-ontology` (accessed 16 June 2016)

etc.) should be stored. The challenge is to define the vocabulary not too broadly or too narrowly.

R2 – Reuse of existing ontologies. Reuse of knowledge is a key concept of the semantic web. This is necessary to avoid the dissemination of terms and concepts with overlapping meaning via different URIs. Moreover, reuse of ontologies saves effort, because one does not need to start development from scratch. This approach also facilitates interoperability between, for instance, multiple sensor networks using the same vocabularies.

R3 – Lightweight. The result should be a lightweight ontology which is easy to (re)use. Users should be able to start working with the terms quickly and without reading detailed documentations. Lightweight ontologies are commonly used for data standardization purposes and when resulting applications do not involve complex reasoning tasks based on so-called heavy semantics.

R4 – Reusability and extensibility. Users of the vocabulary should be able to reuse and extend it easily. This requires that the vocabulary is designed generic enough in order to extend it easily with more concrete definitions, if required.

R5 – Separation of knowledge. The problem of conflating concepts arises when the domain to be modeled is not defined precisely and completely. The vocabulary at hand should avoid conflating terms by separating the used concepts in a way that enables flexible queries along different measurement characteristics. Moreover, we aim to separate domain knowledge from operational knowledge when designing the ontology. Domain knowledge acts as static and stored background knowledge, which supports knowledge extraction. It does not change frequently. Operational knowledge is represented through concrete observations that flow in steadily in the form of data streams. These two layers shall be separated in the model. By preventing conflated concept definitions and ensuring a clear separation of knowledge domains the ontology facilitates semantic interoperability and powerful query mechanisms. It makes the concepts easier to query and to align them to vocabularies of other domains (this is also related to the extensibility requirement R4 above). Moreover, the combination of static knowledge (e.g., geographic maps, point-of-interest data, etc.) with operational knowledge (i.e., dynamic data streams) improves the ability to deduce new knowledge.

R6 – Observation Aggregation. It should be possible to aggregate sensor observation data with typical aggregation functions like MIN, MAX, SUM, or AVG. Moreover, it should be possible to combine and aggregate observations that stem from different data sources and vary in their dimensions. For instance, a query could ask for "*Provide the average temperature of sensors located near Vienna from the last two weeks*". The ontology has to be designed to allow aggregation of data along temporal or spatial dimensions.

R7 – Dynamic integration of data streams. The ontology should support the definition of new data streams with their accompanying data structure. We expect that data streams will be added and removed over time and their encoding in the data model should be flexible enough to account for this. This requires that data streams can be defined by following a given structure in order to ease the process of integrating new sources.

R8 – Exploitation of hierarchical structures. Ontologies can store knowledge in hierarchical structures; an ontology for the environmental domain can make use of this characteristic to support convenient query formulation. A potential query could be "*Provide all sensor observations which measure precipitation properties*". If the observed properties are encoded in a hierarchy, where different types of precipitation such as snowfall or rainfall are subsumed under the same concept, general queries can be formulated conveniently.

R9 – Stream and observation discovery. This requirement is related to R4 and R8; it states that both streams and observations should be discoverable along key dimensions such as spatial, temporal, and environmental characteristics. For example, if a sensor measures *air temperature*, it should be possible to query sensors based on either observed properties (*temperature*) or features (*air*). This allows to process queries regarding either properties or features in a flexible way. Moreover, since we also encode temporal and spatial knowledge, these dimensions should be supported for querying as well.

2.3.2 Design Considerations

Several considerations and design decisions were made while implementing above requirements. They will be described in the next sections.

2.3.2.1 Decisions between ontology alternatives

QUDT vs. SWEET Several ontologies are available to encode environmental data. In the context of our work, we want to model properties of observations, e.g., *temperature*, *length*, *speed*. Two ontologies are available for this purpose: Quantities, Units, Dimensions and Data Types Ontology (QUDT)[9] (prefix: qudt) and Semantic Web for Earth and Environmental Terminology (SWEET)[10] (prefix: sweet). We did a non-exhaustive evaluation of environmental properties. Table 2.2 shows that SWEET (*sweet:Property*) supports a wide range of the properties, whereas QUDT does not support all of them (*qudt:QuantityKind*). QUDT developers state that *height* and *depth* are not encoded as quantity kinds, because they are specific quantities of the quantity kind *length*.

The terms of the SWEET vocabulary are more targeted towards the environmental domain. This and the fact that units of measure are also encoded in SWEET (and related

[9] `http://www.qudt.org/` (accessed 16 June 2016)
[10] `https://sweet.jpl.nasa.gov/` (accessed 16 June 2016)

Properties	QUDT	SWEET
Temperature	Y	Y
Length	Y	Y
Depth	N	Y
Height	N	Y
Pressure	Y	Y
Angle	Y	Y
Direction	N	Y
Speed	Y	Y

Table 2.2: Comparison of supported observation properties of the QUDT and SWEET ontologies.

to *sweet:property*) — which enable intelligent reasoning and querying mechanisms — led us to the decision to use SWEET.

GeoSPARQL vs. WGS84 Several ontologies to encode spatial data are available. The WGS84 ontology[11] is a popular example[12]. Its simplicity, however, comes with the trade-off of being less expressive. In contrast, the GeoSPARQL ontology[13] is an official Open Geospatial Consortium (OGC) standard and allows for more complex spatial encodings such as lines or polygons. The GeoSPARQL ontology enables queries such as "Provide observations from all sensors in the city of Vienna", given the coordinates of Vienna are described as a polygon. GeoSPARQL also comes with a query language that allows, for instance, queries based on spatial relations.

2.3.2.2 Reusing Ontologies

There are two options for ontology reuse: either through the *owl:imports* statement, or through redeclaration of external classes and properties.

Using *owl:imports* makes the entire external ontology a part of the new ontology. This may not always be desired behavior, for instance, if large and complex ontologies are imported. Moreover, *owl:imports* is transitive, that is, if ontology A imports ontology B, and B imports C, then A imports both B and C. Because we aim to design a lightweight ontology (*R3*), we decided to import small ontologies only. In cases where we want to reuse terms of large ontologies, or if we just pick particular terms, we simply redeclare them.

[11] `http://www.w3.org/2003/01/geo/` (accessed 16 June 2016)

[12] Richard Cyganiak runs `prefix.cc`, a website to look up namespace prefixes. He did an evaluation on the most requested prefixes and WGS84 shows up at the 6th place. See `http://richard.cyganiak.de/blog/2011/02/top-100-most-popular-rdf-namespace-prefixes/` (accessed 16 June 2016).

[13] `http://www.geosparql.org/` (accessed 16 June 2016)

Avoiding *owl:imports* for large ontologies has the advantage that semantic web tools such as reasoners do not need to load all referenced ontologies when our vocabulary is used. The redeclarations are useful for Web Ontology Language (OWL) tools which require definitions of used classes and properties. Linked data applications can use the URIs of the definitions for dereferencing. Overall, *owl:imports* is suitable when reuse of all axioms of an external ontology is desired. Given that our goal is to reuse existing and well-known terms to facilitate interoperability, redeclaring them is sufficient.[14]

We use redeclarations of the following classes: *sweet:HumanActivity*, *sweet:Phenomena*, and *sweet:Substance*. These classes are furthermore encoded as subclasses of *ssn:Feature-OfInterest* meaning that the respective SWEET classes are used as features of interest for the actual instance data. Similarly, *sweet:Property* is aligned with *ssn:Property*. From the Time Ontology in OWL we declare *time:Instant* and *time:inXSDDateTime* to capture temporal data, i.e., when an observation has been made. From the GeoSPARQL Ontology we reuse *geo:hasGeometry* and *geo:Geometry* to store spatial information, i.e., lo-cations of sensors. *Qudt:numericValue* is used to encode actual sensor measurements. We use *owl:imports* only for the Data Cube vocabulary and for the SSN ontology, because we reuse large parts of these ontologies.

2.3.2.3 Modeling of Spatial and Temporal Observation Data

The following considerations are made to store spatial (i.e., locations of observations or sensors) and temporal data (i.e., when an observation has been made) with the proposed model.

We initially considered attaching location data, i.e., the location where an observation was made, directly at the observation level. However, to conform with the SSN ontology, we decided to do this at the sensor level via *geo:hasGeometry* (cf. Figure 2.2). This approach is less redundant, because we do not capture any data of moving sensors. In other words, we assume that the location of sensors is static. Hence, there is no reason to attach the same location information for one data stream to each observation again and again. Moreover, the sensor is attached to each observation through *ssn:madeObservation*, meaning that the location of an observation can easily be retrieved.

A similar rationale applies for the encoding of units of measurement. Those are stored at the data set level (*es:DataStream*) via the *sweet:hasUnit* property. Each *es:DataStream* is related to exactly one *qb:DataStructureDefinition*. Therefore, it generates observations of only a single property and the unit of measurement does not change over time. Observations also only observe a single property at a time, hence, there is no need to encode multiple units of measurement for a data stream. The official standardization document of the Data Cube vocabulary also describes a special *qb:componentAttachement*

[14] On semanticweb.com different ways to reuse terms when an ontology is designed have been discussed and explained (cf. `http://answers.semanticweb.com/questions/18505/ontology-import-vs-owlsameas-in-ontology-design` (accessed 16 June 2016))

property which can be used to attach attributes, for instance the unit of measurement, to the whole data set in order to avoid redundancy[15].

Data Cube allows to encode multiple measures at a single observation. For instance, an air sensor could observe temperature and wind speed. The vocabulary provides two mechanisms to implement this, namely multi-measure observations and measure dimensions. However, because we want to comply to both the SSN ontology and Data Cube and because it is not feasible to create observations based on the SSN ontology which capture multiple observed properties, we decide to model single property observations. If we would follow one of the approaches proposed in the Data Cube specification (multi-measure and measure dimension observations), this would make our vocabulary incompatible with SSNO.

2.3.2.4 Defining the Measure Property according to the Data Cube vocabulary

The Data Cube vocabulary prescribes the definition of a Data Structure Definition (DSD). The actual observation data is stored according to the structure defined in the DSD. We have to trade off interoperability and flexibility when the Data Cube vocabulary and the SSN ontology are combined.

In the DSD the data provider uses *qb:measure* to create a relationship to a class of type *qb:Measure-Property* which in the observation data will materialize as a relation to the actual measured value. For instance *eg:CO2* may be declared as a *qb:MeasureProperty*. An example observation will then, e.g., contain the triple *eg:Observation eg:CO2* "114". However, to comply to both vocabularies, we declare *ssn:observationResult* as a *qb:Measure-Property* in the DSD. Starting from an observation, the actual measurement can then be queried via the classes *ssn:SensorOutput* and *ssn:ObservationValue*.

2.3.3 Reused Ontologies

Table 2.3 summarizes the used vocabularies, their namespaces, prefixes, application domain and how they were reused (either via *owl:imports* statement (I) or via redeclaration (R)). The prefixes in the table are also used further in this chapter to describe classes and properties of respective ontologies. Furthermore, we introduce the prefix *es* (environmental streams) which is the namespace of the developed ontology. It defines new classes deduced from upper ontologies and reuses external properties. Figure 2.1 depicts how existing ontologies were combined via subclass statements.

Figure 2.2 provides an overview of the classes and their relations via properties from an SSN ontology-centric point of view, whereas Figure 2.3 provides an overview of used classes and relationships from a Data Cube vocabulary-centric point of view.

[15]There has been some discussion about how and if multiple properties can be encoded into a single observation of the SSN ontology which influenced our decision to also only create single property observations (cf. `http://lists.w3.org/Archives/Public/public-xg-ssn/2014Apr/0007.html` (accessed 16 June 2016)).

Ontology Name	Namespace	Prefix	Domain	Reuse
Semantic Web for Earth and Environmental,Terminology (SWEET)	http://sweet.jpl.nasa.gov/2.3/	sweet	Environmental terms	R
Semantic Sensor Network Ontology (SSNO)	http://purl.oclc.org/NET/ssnx/ssn#	ssn	Sensor and Observation descriptions	I
RDF Data Cube Vocabulary (QB)	http://purl.org/linked-data/cube#	qb	Multi-dimensional observations	I
Time Ontology in OWL	http://www.w3.org/2006/time#	time	Temporal data	R
GeoSPARQL	http://www.opengis.net/ont/geosparql#	geo	Spatial data	R

Table 2.3: Reused ontologies

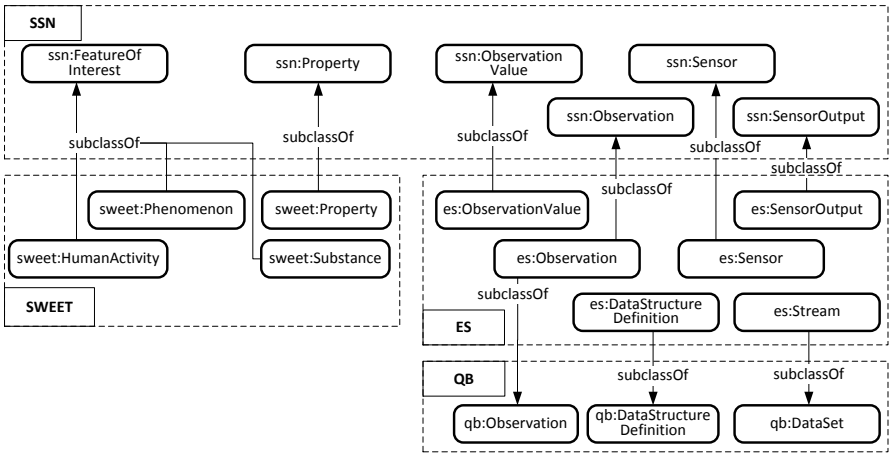


Figure 2.1: Reuse and combination of external ontologies via *owl:subclassOf* properties. *Es* is the prefix of the developed environmental streams ontology.

In the former, it is shown that we derive *es:Sensor*, *es:Observation*, *es:SensorOutput*, and *es:ObservationValue* from respective classes of the SSN ontology and the Data Cube vocabulary. *Sweet:HumanActivity*, *sweet:Phenomena*, and *sweet:Substance* are aligned with *ssn:FeatureOfInterest*, since they contain conceptual domain knowledge which later will be encoded as Features complying with SSN ontology best practices. Furthermore, we use *geo:hasGeometry* in combination with *geo:Point* to model sensor locations and *ssn:observationResultTime* in combination with *time:Instant* to capture temporal information of observations.

Figure 2.3 shows that *es:Stream* is derived from *qb:DataSet*. We see that units of measurement are encoded as *sweet:Unit* via *sweet:hasUnit* at the data set level. The bottom half of the figure represents instance data, i.e., the actual data structure definition, observation and data stream instances according to the Data Cube vocabulary.

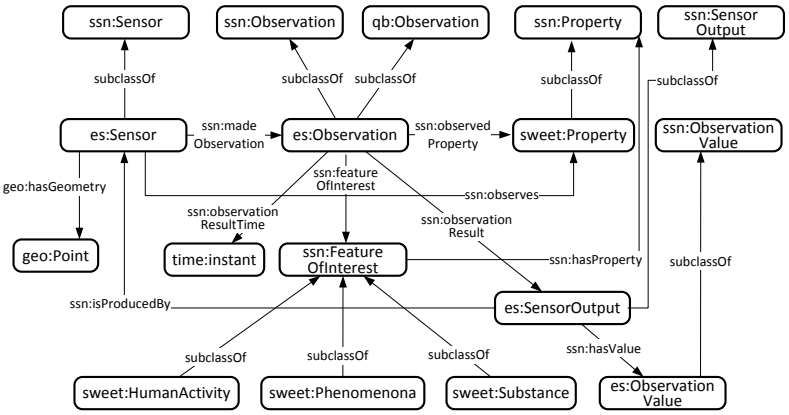


Figure 2.2: SSN ontology-centric view over classes and their relations

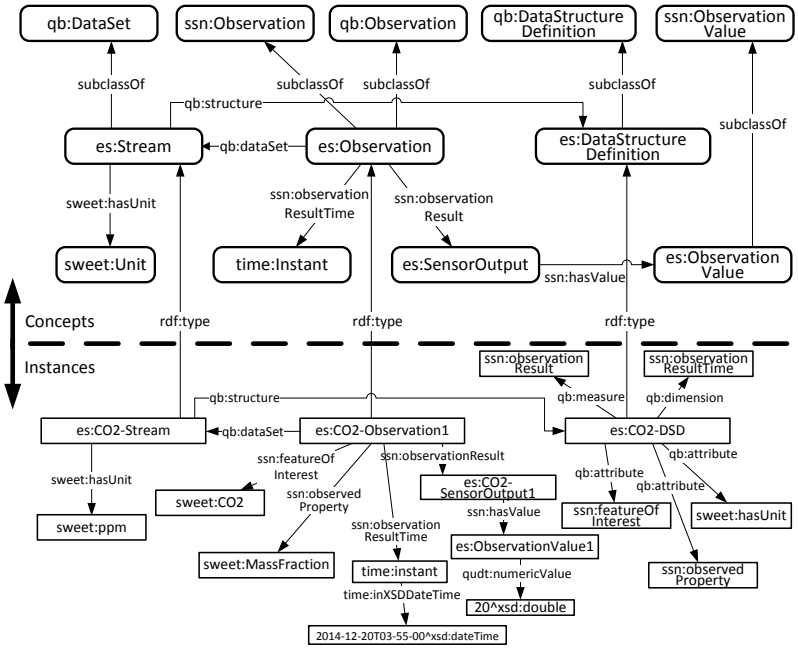


Figure 2.3: Data Cube vocabulary-centric view over classes and their relations[16]

[16]On the bottom half instance data is visualized, i.e., how an actual data stream, observation and data structure definition is modeled based on the created vocabulary. Note the absence of location data (*geo:hasGeometry*) as a property of the observation and the absence of the unit encoding as a property of the data stream instance (*es:CO2-Stream*). For details cf. Section 2.3.2.3.

2.4 Architecture

In this section we describe the relation between the vocabulary and our architecture and its three different stages, i.e., *Data Acquisition*, *Data Transformation*, and *Data Streaming* as shown in Figure 2.4. The figure also shows to which part of the architecture each research question relates to.

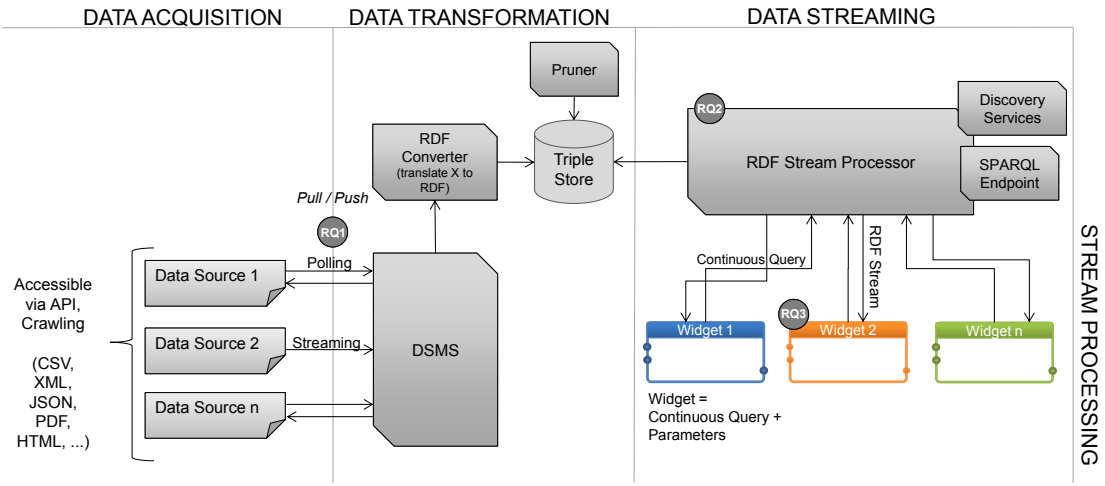


Figure 2.4: Architecture of the proposed approach

2.4.1 Data Acquisition and Data Transformation

Environmental data is available from different repositories, each providing unstructured, semi-structured, and structured data. In many cases, data is presented only on a webpage or exposed via non-standardized interfaces. To allow for timely provision of data via our platform, such data has to be retrieved on a regular basis. Data available in (semi)structured formats is more straight-forward to handle, but needs to be converted into JavaScript Object Notation for Linked Data (JSON-LD), a recent W3C recommendation [LSK14] that we use as our internal data exchange format.

After conversion, the data is fed into a Data Stream Management System (DSMS) and the triples are stored in a triple store. The RDF converter uses domain ontologies to enrich incoming data sources with semantic knowledge, which later will be utilized to support features such as stream processing or contextualized sensor discovery. The DSMS is dependent upon the RDF stream processor implementation. We conducted a thorough evaluation with a novel benchmark which we present in Section 3 to identify the most suitable stream processor based on the requirements of our scenario.

Data sources differ in type, access mechanisms, and semantic meaning. Ontologies have been used for years as a means to overcome the resulting heterogeneity [NM01, Hul97,

Gru93]. In the context of our proposed framework they are a valuable tool to define a comprehensive and standardized semantic model which is a prerequisite for semantic search and knowledge extraction from sensor-generated data.

Furthermore, differences in number and range of observed properties as well as update frequency (varying from stream data, i.e., real time updated data, to hourly updated data) result in large variation in the amounts of data provided, which has to be taken into account when we evaluate implementation candidates for the RDF converter.

2.4.2 Data Streaming

The first stage of data flow in the architecture, i.e., data acquisition, results in semantically annotated observation data, i.e., RDF streams that can be presented to end-users. In the second stage, we provide real time data to the user.

We make use of the publish-subscribe design pattern, which controls what messages are sent by entities that publish data to receiving entities [EFGK03]. In the context of the proposed framework, loosely coupled widgets can act as publishers and subscribers. A key advantage of this approach is that through parallel operations, message caching, and routing, this pattern provides the scalability needed to handle flexible stream compositions on our platform. Consequently, it addresses the first challenge of providing environmental data streams to users by allowing clients to subscribe to data streams dynamically.

Furthermore, because of the continuity and potentially large size of data streams, storage is a key issue. To avoid bottlenecks in subsequent procedural steps, we need to define when data becomes outdated and can be deleted. In case historical analysis is required, techniques to aggregate outdated data in order to save storage capacity need to be implemented.

The architecture supports flexible exploration of the data. Users can combine widgets to answer questions related to environmental data. Via drag-and-drop, these widgets can be combined into mashups. A mashup can satisfy information needs, e.g., by displaying points of interest that satisfy certain air quality criteria. Widgets leverage the modeled semantics and can be combined in many different ways.

We apply stream reasoning techniques provided through SPARQL extensions, i.e., windowing functions and federation of static data with dynamic streams. Each corresponding data stream is represented by a widget. A web-based graphical interface allows users to assemble these widgets and set parameters for their processing functions. In doing so, users can efficiently explore data streams.

These processing widgets have encoded queries based on stream-specific criteria, e.g., time windows or aggregates (*sum*, *count*, *average*, etc.), and return RDF triples that answer this query, ultimately allowing ad hoc combination of data streams with static or streaming data. Presentation widgets provide mechanisms to visualize output via, for instance, maps, bar charts, line charts, pie charts, and histograms. This step covers data

analysis via continuous stream queries, publication via RDF graphs, and visualization via presentation interfaces.

2.5 Summary

In this section, we propose a data model that serves as a basis for a widget-based framework. This framework enables the exploration of environmental data streams in an urban context. We divide the architecture into three stages and identify key issues that need to be addressed. These include the definition of a new vocabulary for environmental stream data deduced from already existing and well-adopted ontologies, and applying semantic stream processing methods to facilitate reasoning.

We evaluate potential ontologies for reuse. The resulting vocabulary aligns and harmonizes terms of both the Semantic Sensor Network Ontology and RDF Data Cube vocabulary. This enables interoperability and allows to reuse recommended query patterns and best practices which are defined for both ontologies.

We identified nine high-level requirements based on the domain of environmental stream data integration which were addressed as follows.

R1 The data integration requirement is achieved via the definition of a conceptualization in terms of an ontology which combines domain ontologies into a unifying framework.

R2 We reuse existing ontologies to avoid formulating new concepts which have already been defined before. This also allows users of these ontologies to adapt their queries without having to introduce new concepts.

R3 The requirement to provide a lightweight solution in terms of complexity is satisfied. The developed vocabulary consists of 20 classes and 13 properties. It does not support complex reasoning tasks, but facilitates to build applications which need to process environmental stream data based on a common data model.

R4 Reusability and extensibility are ensured by allowing to use terms of external vocabularies, e.g., terms of the SWEET ontology which comprises an extensive collection of environmental terms. Moreover, the vocabulary can be extended and linked via common methods, such as subclassing.

R5 Separation of knowledge is achieved by splitting operational knowledge and domain knowledge in the model. We also provide a definition of sensor observations in order to allow for granular queries by separating concepts into observed properties, observed features, and units of measure.

R6 The design allows to aggregate observations along temporal and spatial dimensions via using the stored metadata of sensors and their observations.

R7 We follow the provided structure of the vocabulary, i.e., we define the core concepts of the ontology. Thereby, we satisfy the requirement to enable dynamic integration of new data streams.

R8 Exploitation of hierarchical structures is satisfied by linking the vocabulary with the SWEET ontology. In this ontology, the concepts are arranged in a strict hierarchical scheme than can be used to formulate queries. These queries subsume concepts belonging to the same branch of the hierarchy.

R9 The discovery of streams and observations is provided via the foundational design and structure of the vocabulary. Streams and observations are defined as separate concepts and linked together, hence, allowing to be discovered separately.

Our architecture aims to facilitate access to and reuse of public environmental data sources. At present, most of these sources provide only infrequent snapshots of static data. Given the ongoing efforts in the area of Open Data, we expect that in future more real time sources will be available which will facilitate innovative applications in the environmental domain. In the long term, the proposed system could serve as an open data platform for citizens of a "smart city". Applications built on top of this architecture can bring together developers and users. For each of them, it should be as easy as possible to create, (re)use, modify, and execute mashups. By overcoming technical barriers of adoption, citizens will be enabled to interact with the available data sources, e.g., stream data, open data, linked data, tabular data while accessing data in different formats. Creative (re)combination of available data enables the creation of new knowledge. The vision is to provide a platform for dynamically building applications that leverage semantically enriched environmental data in a timely manner. Ultimately, this could lead to a better understanding of the environment in the local context of a city.

In this chapter we have designed an ontology based on a set of requirements for environmental stream data modeling. The presented requirements generally also apply to other stream data modeling settings besides the environmental domain. Therefore, they can be reused to facilitate the design of similar ontologies for other domains, such as IoT or health care. We validated the resulting ontology by discussing it with respect to each requirement.

CHAPTER 3

Evaluating Stream Processing Engines

This chapter presents our contributions in the field of evaluating stream processing engines. Initially, we discuss related work in semantic stream processing (Section 3.2). Next, we present a novel framework to benchmark RDF Stream Processing (RSP) engines called YABench (Section 3.3). Next, we validate our approach through a comparison with results from previous benchmarking efforts, namely CSRBench (Section 3.4). Next, we discuss our experimental setup (Section 3.5) and present the results of our analysis (Section 3.6). The results provide evidence that semantic stream processing engines suffer from performance and correctness issues. Finally, we conclude with a discussion on future implications and an outlook on future work (Section 3.7).

3.1 Motivation

Since environmental data is generated by continuous sensor measurements, their unified processing requires a shift from a static and persistent towards a continuous paradigm. In particular, applications which enable decision-making based on such data foster the need for efficient processing of and reasoning over dynamic data. Making sense of frequently changing data flows to draw timely conclusions about the state of our environment is crucial. This requires proper means to consider the temporal dimension of the data.

We evaluate the use of continuous queries [ABW06] for dynamic data processing in the environmental domain. More precisely, we compare different approaches which have already been proposed in the domain of semantic stream processing with respect to environmental stream data based on our initially defined challenges (cf. Section 3.2).

A number of semantic stream processing engines — also known as RSP engines — have been proposed that provide (limited) capabilities for reasoning and to cope with hetero-

geneity. Prominent examples include Continuous SPARQL (C-SPARQL) [BBC+10b], Continuous Query Evaluation over Linked Streams (CQELS) [LDPH11], and SPARQL$_{\text{Stream}}$ [CCG10]. They can provide a basis for innovative applications in data stream analytics, but there is currently no consensus on how these RSP engines should be benchmarked. Especially performing evaluations with respect to the challenges posed by environmental stream data, is currently not offered by available benchmarking frameworks. A common framework that can gather experimental results for environmental scenarios and uncover issues in current implementations is therefore necessary.

Several approaches to enable SPARQL-like data access on flows of incoming data have been proposed recently. Most of them operate over *windows* to limit an unbounded input stream from which a finite amount of elements is selected via queries. Conversely, there are approaches which see flowing information as (complex) *events* whose occurrence is used to detect patterns. The present work deals with window-based stream processing engines. Differences in the implementations of these efforts result in divergent performance-characteristics, which were investigated in previous work [LDP+12, ZDCC12, DCB+13]. Beyond that, they also crucially differ in their understanding and execution of operational semantics. Previous work has highlighted differences in report strategies, query semantics, output operators, and relation notifications. Even though these variations make it difficult to compare engines, such comparisons are crucial to get a better understanding of their behavior. Previously, only isolated aspects such as functional coverage, performance, and correctness, were evaluated through specialized benchmarks, this work compares RSP engines along all of these dimensions.

Motivated by these needs, we present Yet Another RDF Stream Processing Benchmark (YABench), an integrated framework to assess correctness and performance of RSP engines. YABench generates test data, allows for the definition of test scenarios, and provides analyses of the evaluation runs. We put a strong emphasis on reproducibility and visual presentation of results to foster an understanding of the individual characteristics of an engine, including correctness under varying circumstances such as different input loads, window sizes, and window frequencies.

3.2 Related Work

Related work in the field of semantic stream processing can be split into two main parts. First, relevant works for stream processing in general and stream processing in the semantic web will be presented. Stream processing is a relevant area for the presented work, because real time environmental data necessitate processing in a continuous manner. Hence, we can deduce the following challenges: (i) the ability to cope with frequently changing data (*C1*), (ii) the need for reactive and timely answers (*C2*), (iii) the need for precise answers (*C3*), and (iv) the ability to integrate stream data with static data (*C4*). Second, research towards benchmarking stream processing proposals will be discussed. Benchmarking of stream processing is important in order to ensure that a system can satisfy the defined challenges.

3.2.1 Stream Processing Engines

Given the abundant availability of data produced by environmental sensors, processing continuous flows of data has gained major interest [HM06]. Stream processing engines are "capable of timely processing large amounts of information as it flows from the peripheral to the center of the system" [CM12] and therefore suitable to deal with such data.

The two concepts of *timeliness* and *flow processing* are essential for the definition of this new class of systems. Traditional Database Management Systems (DBMSs) (i) require data to be persistently stored before being processed, and (ii) process data only when explicitly queried by users or systems. Both characteristics do not apply in the context of systems processing flows of incoming data. As a consequence, a new model, namely the *data stream processing* model, was proposed [BBD+02].

DSMSs are strongly influenced by traditional DBMS. They typically represent data in a relational model and express queries in declarative languages such as Structured Query Language (SQL). Hence, they are an extension of database systems that provides continuous query answers for constantly changing and unbounded input data. To select recent elements, DSMSs isolate parts of incoming unbounded data streams using time-based or count-based windows [GÖ03]. After this windowing, so-called Relation-to-Stream operators are used to convert resulting tuples into a stream [ABB+04].

The main differences between DSMSs and traditional DBMSs are as follows [CM12]:

- streams are typically unbounded,
- data may not arrive in order, and
- streams have to be dealt with in a one-time processing manner.

A key characteristic of DSMSs is that queries are registered once and then evaluated continuously or periodically. To date, a large number of DSMSs have been developed in the academic literature [CCD+03, BTW+06, LPT99, CDTW00, CJSS03, CGJ+02, ABB+04, ABW06, AAB+05, ACÇ+03a, ACÇ+03b] and various commercial implementations are available [ibm, tib, mic].

Semantic web technologies are often used in data integration scenarios, yet, until recently, they were only applied to static and infrequently changing data. As of late, however, the idea to merge the DSMS paradigm with concepts of semantically enriched data processing techniques has gained momentum. Transferring ideas for semantic data integration to data streaming scenarios allows (i) to fuse static and stream data on-the-fly, (ii) to integrate multiple heterogeneous streams based on elaborate domain models, and (iii) to reason and perform complex queries over combinations of different sources. Selected developments in this field are described in the following.

3.2.1.1 C-SPARQL

Early contributions aimed to formalize a continuous query language for RDF data streams built upon SPARQL [BBC+10c, BBCG10, BBC+10b]. The extended language called C-SPARQL supports timestamped RDF triples and evaluation of continuous queries. Background knowledge and streaming knowledge can be combined within queries. Each query has a fixed evaluation frequency, which decouples query evaluation from the arrival of new data to the stream. In the context of real time requirements, this poses a significant limitation. Moreover, temporal patterns over input elements are not supported.

Figure 3.1 provides a high-level view of the C-SPARQL architecture. Queries can be registered and run continuously at the system. To this end, the engine uses two sub-components, a DSMS and a SPARQL Engine. The first is used to execute continuous queries over RDF Streams, producing temporal RDF snapshots, while the latter retrieves a snapshot as input and runs a standard SPARQL query, producing a quasi-continuous result. This result can be either a stream of variable bindings (for `SELECT` or `ASK` queries) or an RDF stream (for `CONSTRUCT` queries). As implementation of the DSMS C-SPARQL uses Esper[1]. The SPARQL engine is Apache Jena-ARQ[2].

C-SPARQL has been applied in cloud monitoring applications [MBH+13], self-adaptation for cloud environments [DPS13], social listening of events [BDVD+13], and data mining

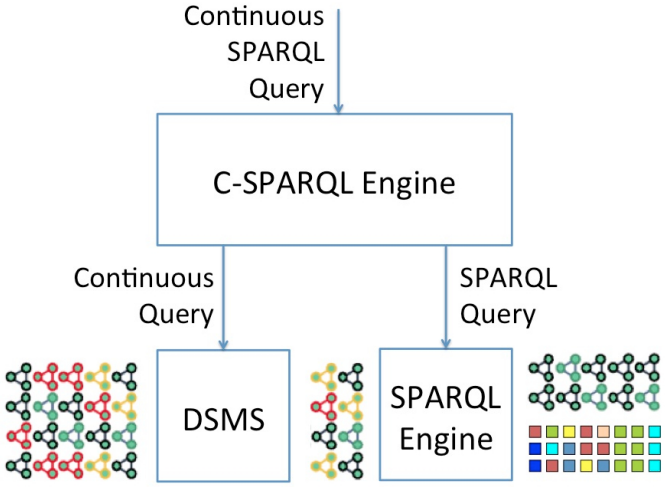


Figure 3.1: High-level architecture of C-SPARQL[3]. The output of the engine can be either an RDF stream (triples depicted as triangles) or a stream of relations (colored squares).

[1] `http://www.espertech.com/products/esper.php` (accessed 16 June 2016)
[2] `https://jena.apache.org/documentation/query/` (accessed 16 June 2016)
[3] Adapted from `https://www.w3.org/community/rsp/wiki/RDF_Stream_Processors_Implementation#C-SPARQL_and_C-SPARQL_Engine` (accessed 16 June 2016)

on micro-post streams [BCD+14]. The engine is still under active development; its code resides at GitHub[4].

3.2.1.2 CQELS

Similar to C-SPARQL, CQELS [LDPH11] also extends SPARQL for data streams. However, there are some notable differences: Rather than evaluating queries periodically, matching triples are reported immediately when they arrive. Therefore, a query will never report duplicate items in its results. Preliminary experiments suggest significant performance advantages of CQELS over C-SPARQL [LDP+12]. Recent experiments [DBDV13, DCB+13], however, highlight essential differences in the semantics of these two engines.

In contrast to the C-SPARQL Engine which uses a "black box" approach and delegates the processing to other engines, CQELS uses a "white box" approach (cf. Figure 3.2). This means that required query operators are implemented natively to avoid processing overhead and limitations of closed system regimes. CQELS provides a flexible query execution framework with the query processor dynamically adapting to changes in the input data. To improve query execution with regard to delay and complexity, query operators are continuously reordered. Moreover, retrieval of large linked data collections is minimized by caching of intermediate query results. CQELS has been used in the area of urban data streams [GAM14] and mashups [QSLPH12].

3.2.1.3 SPARQL$_{Stream}$ and Morph-streams processor

SPARQL$_{Stream}$ [CCG10] is an extension to SPARQL that supports all Relation-to-Stream operators, i.e., Rstream, Istream, and Dstream. These operators are applied to control

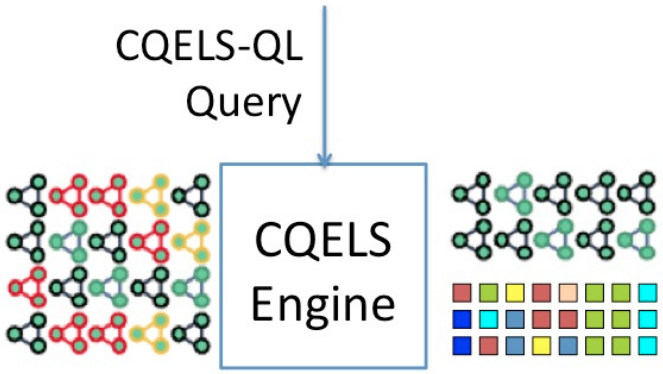


Figure 3.2: High-level architecture of CQELS[5]. The output of the engine can be either an RDF stream (triples depicted as triangles) or a stream of relations (colored squares).

[4] https://github.com/streamreasoning/CSPARQL-engine (accessed 16 June 2016)

[5] Adapted from https://www.w3.org/community/rsp/wiki/RDF_Stream_Processors_Implementation#CQELS_and_CQELS-QL (accessed 16 June 2016)

the output of queries which can either contain only new data, deleted data, or all data [MWA+03]. Moreover, $\text{SPARQL}_{\text{Stream}}$ supports ontology-based access to legacy data stream sources. Its queries are translated into target languages such as Esper, SNEE [GBG+11], GSN [AHS06], and Xively[6]. Morph-streams is an RDF stream query processor for the continuous execution of $\text{SPARQL}_{\text{Stream}}$ queries against virtual RDF streams.

Figure 3.3 shows Morph-streams that uses RDB to RDF Mapping Language (R2RML) [DCS12] to define mappings between ontologies and data streams. $\text{SPARQL}_{\text{Stream}}$ queries are rewritten to continuous queries depending on the target language of the underlying DSMSs. Essentially, a $\text{SPARQL}_{\text{Stream}}$ query is represented as relational algebra expressions extended with time window constructs. This allows to perform logical optimization and to translate the algebraic representation into a target language (SNEE and ESPER) or Representational State Transfer (REST) Application Programming Interface (API) request (GSN and Xively). This proposed SPARQL extension is used in the domain of sensor networks to query observation data [CCJA12].

3.2.1.4 Complex Event Processing

In contrast to DSMSs, researchers also developed approaches aimed to detect complex patterns in data streams. This paradigm is called Complex Event Processing (CEP). In CEP-based systems, incoming data items represent timestamped notifications of event occurrences. These can include both instance data (e.g., sensor reading values)

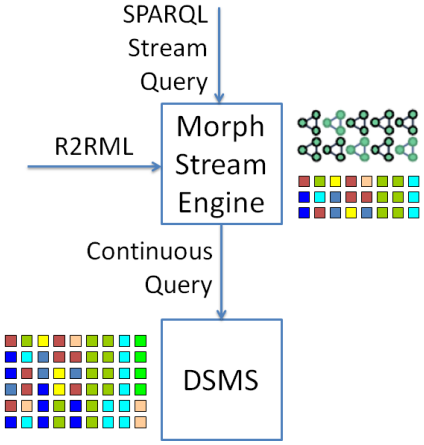


Figure 3.3: High-level architecture of $\text{SPARQL}_{\text{Stream}}$[7]. The output of the engine can be either an RDF stream (triples depicted as triangles) or a stream of relations (colored squares).

[6] https://xively.com/ (accessed 16 June 2016)

[7] Adapted from https://www.w3.org/community/rsp/wiki/RDF_Stream_Processors_Implementation#SPARQLstream_and_Morph-streams_processor (accessed 16 June 2016)

and metadata (e.g., location of where the reading was taken). CEP systems are based on the concept of creating higher-level composite events from primitive events via processing rules [MUHB14, Luc01]. Semantic CEP systems are able to process RDF data include Event Processing SPARQL (EP-SPARQL) [AFRS11], INSTANS [RNT12], and Sparkwave [KCF12]. However, given that we do not aim to deduce complex composite events based on granular measurements, we do not include a more detailed discussion here.

3.2.1.5 Summary

Table 3.1 summarizes the currently available DSMS-based semantic stream processing engines with respect to our requirements. We also map the requirements represented in columns to the initially stated challenges which we identify with respect to processing of environmental data (cf. Section 3.2). We see that all challenges can be dealt with by the discussed systems.

We observe that *continuous queries*, integration of *background knowledge*, use of *combined streams* in single queries, and *S2R operators* are well supported. *Temporal operators* in queries, such as *before*, *after*, or *during*, and *reasoning support*, by contrast, are largely unsupported. We provide more details on each requirement and how it relates to the domain of environmental data streams in the following.

Time model Timing of events is crucial in the environmental domain and a suitable time model is required to establish temporal relations between sensor observations. We distinguish between *interval* (two timestamps establish lower and upper bounds of an interval), *single timestamp* (single timestamps represent a single point in time), and *implicit* (no explicit timestamps, sequence order only) time models.

Continuous queries Continuous access to data is a key requirement for the filtering of high volumes of environmental data streams and for subsequent real time monitoring and reasoning. These queries provide the foundation to create timely responses.

Engine	**Time model**	**Continuous queries**	**Background knowledge**	**Combined streams**	**Temporal operators**	**Reasoning support**	**S2R operators**
Environmental data challenges	*C2*	*C2, C3*	*C4*	*C4*	*C4*	*C3*	*C1, C2*
C-SPARQL	Single Timestamp	✓	✓	✓	-	RDF entailment	✓
CQELS	Single Timestamp	✓	✓	✓	-	-	✓
$\text{SPARQL}_{\text{Stream}}$	Single Timestamp	✓	-	✓	-	-	✓

Table 3.1: Semantic stream processing requirements and challenges[8]

[8] **S2R** denotes *Stream-to-Relation* or *window* operators (see [ABW06]).

Background knowledge The ability to integrate background knowledge into results is important in the environmental domain, where metadata about the sensor network is used to interpret measurements correctly. The requirement indicates if an engine can integrate static knowledge with streamed knowledge, which facilitates contextual interpretation of observed events. For instance, sensor locations, observed features, and observed properties are organized into static repositories (e.g., ontologies). In order to draw relevant and correct conclusions, being able to combine such background information with incoming streams of events is crucial.

Combined streams It is necessary to allow for queries that operate on multiple incoming data streams. In our application domain, input streams originate from diverse sources, hence, the ability to refer to different streams within a single query is an important feature.

Temporal operators Enabling the identification of event patterns based on complex temporal specifications via special operators is sometimes required when dealing with environmental data. For instance, to identify causal relationships between events, such as *increased air pollution* and *car traffic*, temporal operators are essential. Elementary events can occur before, during, or simultaneously with other events.

Reasoning support facilitates the inference of new knowledge which is not explicitly stated in data streams. Our preliminary evaluation shows that only RDF entailment (*rdfs:domain*, *rdfs:range*, *rdfs:subPropertyOf*, *rdfs:subClassOf*) is available. In the environmental domain, these entailments enable to use the structure of ontologies to compose more complex queries which, for instance, group classes of sensors observing similar properties together.

S2R operators enable the partitioning of incoming unbounded flows of data into bounded partitions based on time- or tuple-based windows. Flexible windowing mechanisms are useful for computationally efficient processing of dynamic data streams.

3.2.2 Stream Processing Benchmarks

The preliminary results shown in Table 3.1 suggest that the engines provide fair coverage of the requirements shown in the columns. However, in order to get a more thorough understanding with respect to environmental data streams and to be able to quantify the performance in more detail, we need to benchmark the respective engines.

In the traditional data streaming domain, the Linear Road benchmark [ACG+04] is widely used to evaluate DSMSs. It is based on a toll system simulation where parameters such as traffic congestion and accident proximity influence toll calculation. The benchmark consists of a historical data generator, a traffic simulator, a data driver, and a validator. Linear Road provides a comprehensive framework for experiments and enables performance comparison between DSMSs and alternative systems, such as relational

databases. The authors introduce the *L-Rating* which is a measure of processable input while the DSMS is satisfying response time and correctness requirements.

In the semantic web domain, there are several well-established benchmarks, including Lehigh University Benchmark (LUBM) [GPH05], FedBench [SGH+11], Berlin SPARQL Benchmark (BSBM) [BS09], and DBpedia SPARQL Benchmark (DBPSP) [MLAN11]. In contrast to DSMS benchmarks, they operate on static knowledge bases and focus on performance characteristics such as query execution and load times. They do not address the additional requirements that arise in a streaming context. For instance, they do not cover aspects such as correctness of results under high load or influence of window size on the output. The inconsistent interpretation of the operational semantics of the streaming operators by RSP engines poses additional challenges and makes reuse of existing benchmarks for non-streaming scenarios infeasible.

The RSP research community has also developed a number of specialized benchmarks which will be discussed in detail as follows.

3.2.2.1 LSBench

Linked Stream Benchmark (LSBench) [LDP+12] first allowed comparisons between RSP engines. Based on a social network scenario, the benchmark uncovers conceptual and technical differences between CQELS, C-SPARQL, and Java Event TrAnsaction Logic Inference System (JTALIS)[9]. Furthermore, it highlights differences in performance between these engines and includes test of functionality and correctness of results.

The data schema used by LSBench extends the schema from the Social Network Intelligence Benchmark[10] and is shown in Figure 3.4. The authors use social network data which consists of two layers, one for *stream data* (user activity, such as followings or likes) and one for *static data* (profile data, relationships between users, etc.).

The authors evaluate the query expressiveness of the engines. They propose twelve queries[11] which cover different features and aspects of streaming scenarios. They identify a lack of features, such as negation or nested aggregation. Moreover, the authors identify significant differences in the outputs of engines, even for queries with the same meaning. This suggests that the underlying implementations of the engines differ and that there is a need to find an explanation for the hidden operational semantics.

LSBench introduces a *mismatch* function, which is used to quantify the output (dis)agreement of two engines. For simple queries, the mismatch between the engines is small. However, for more complex queries C-SPARQL misses answers which appear in the results of CQELS. JTALIS also shows strong deviations from the output of C-SPARQL and CQELS

[9] JTALIS is a Java wrapper for ETALIS which is an event processing system used to evaluate EP-SPARQL.

[10] `https://www.w3.org/wiki/Social_Network_Intelligence_BenchMark` (accessed 16 June 2016)

[11] `https://code.google.com/archive/p/lsbench/wikis/queries_list.wiki` (accessed 16 June 2016)

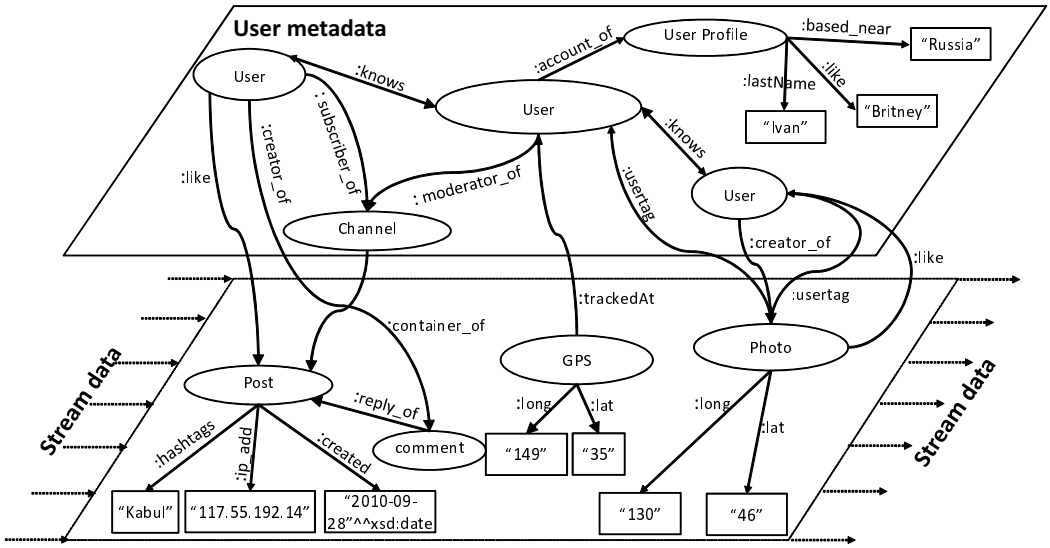


Figure 3.4: Data schema used in LSBench[12]

due to the differences in execution speed. Despite having proposed the *mismatch* function, it is not possible with LSBench to infer which output is correct, and hence no absolute correctness figures are provided.

The performance and scalability tests suggest that the throughput of C-SPARQL is considerably below that of JTALIS or CQELS. The authors of [LDP+12] suggest that (i) an *incremental computing* approach [GÖ03], and (ii) optimization of underlying systems which are used by these engines may increase their performance. The benchmark also reveals that C-SPARQL and JTALIS struggle to cope with big static data sets and suggests to precompute and index intermediate results over static data in order to avoid unnecessary and resource-intense recomputations.

3.2.2.2 SRBench

Streaming RDF/SPARQL Benchmark (SRBench) [ZDCC12] provides a set of queries that cover important RSP-specific aspects such as joining static data with stream data, or performing ontology-based reasoning. The benchmark is "the first general-purpose benchmark that is primarily designed to compare stRS [streaming RDF/SPARQL] engines". Its authors conduct a functional evaluation of C-SPARQL, CQELS, and $\text{SPARQL}_{\text{Stream}}$ and conclude that the capability of these engines is still limited with respect to support for SPARQL 1.1 features, `ASK` queries, relation-to-stream operators, and reasoning.

[12] Adapted from [LDP+12]

SRBench uses LinkedSensorData [PHS10] which is a real-world linked data set containing US weather data published by Kno.e.sis[13]. To check the capability of an engine to answer queries over interlinked data sets, the authors additionally use the GeoNames[14] and DBpedia [ABK+07] data sets. An overview of the used data sets and their interrelations is shown in Figure 3.5.

SRBench provides a set of 17 queries[15] covering a broad spectrum of functional aspects with respect to stream data processing. Because no standard query language for stream data exists, the queries are first presented by means of a descriptive definition. Based on these definitions, the actual queries for each targeted system (CQELS, C-SPARQL, and $\text{SPARQL}_{\text{Stream}}$) are formulated.

At the time of the evaluation, all three engines had strong limitations especially with respect to SPARQL 1.1 features. `ASK` queries are not supported, and $\text{SPARQL}_{\text{Stream}}$ was the only engine that implemented the window-to-stream operator `DSTREAM`. Reasoning, another key distinguishing aspect of RSP engines, was not — and still is not — supported. At present, only C-SPARQL can perform reasoning based on simple RDF entailment. Due to the focus on functional evaluation, SRBench does not recognize the different operational semantics of the benchmarked systems. However, the authors of SRBench propose correctness metrics such as precision and recall in order to validate query results.

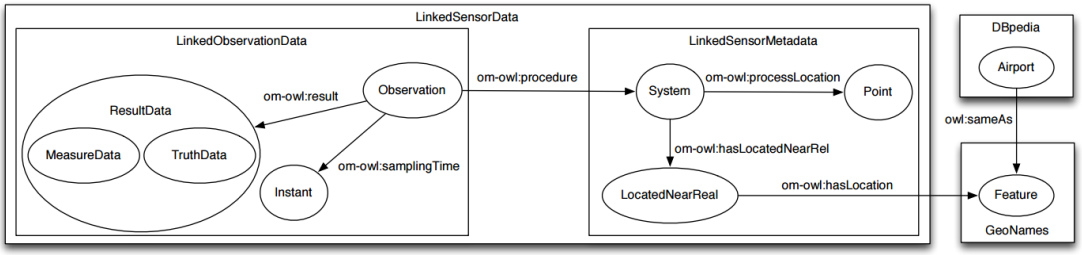


Figure 3.5: Data sets used in SRBench and their interrelations[16]

3.2.2.3 CSRBench

In Correctness checking Benchmark for Streaming RDF/SPARQL (CSRBench) the authors check if stream processing engines produce correct results. Compared to previous benchmarks, performance characteristics and functional coverage are ignored. The main goal, hence, is to describe the *operational semantics* of a generic stream processing engine and to then use this abstract definition to characterize and interpret the functionality of existing approaches. The operational semantics essentially are a description of *how* an engine works and interprets its operators. To this end, the authors of CSRBench use the Scope, Content, Report, and Tick (SECRET) model [BDD+10].

[13] `http://knoesis.wright.edu/` (accessed 16 June 2016)
[14] `http://www.geonames.org/` (accessed 16 June 2016)
[15] `https://www.w3.org/wiki/SRBench` (accessed 16 June 2016)
[16] Adapted from [ZDCC12]

CSRBench [DCB+13] focuses on evaluating a system's compliance to its respective operational semantics. This benchmark is complementary to functional (SRBench) and performance-based (LSBench) evaluations. The authors find that none of the tested engines passes all tests and provide a detailed account on why certain engines fail at specific queries. Whereas this is a first step towards correctness validation of RSP engines, there is a lack of more detailed results evaluating correctness of engines over time. CSRBench introduces an oracle that computes a binary metric (correct or incorrect) to determine the validity of query output.

Table 3.2 shows the main results of CSRBench. For queries *Q1*, *Q2*, and *Q3* it can be concluded that all engines produce the output which is expected according to their operational semantics. The output between the engines also only varies slightly. $\text{SPARQL}_{\text{Stream}}$ does not produce any output, if there are no matching triple patterns, whereas C-SPARQL produces an empty output result set.

Query	Query Type	C-SPARQL	CQELS	$\text{SPARQL}_{\text{Stream}}$
Q1	`SELECT` (single observations) + `FILTER`	✓	✓	✓
Q2	`SELECT` (single observations) + `FILTER`	✓	✓	✓
Q3	`SELECT` (single observations) + `FILTER`	✓	✓	✓
Q4	`SELECT` (aggregate observations) + `FILTER`	✓	×	×
Q5	`SELECT` (single observations) + `FILTER`	×	✓	✓
Q6	`SELECT` (join of different observations) + `FILTER`	✓	×	✓
Q7	`SELECT` (join of different observations) + `FILTER`	✓	×	✓

Table 3.2: Results of CSRBench showing correctness of results for different types of queries[17]

For the remaining queries (*Q4* to *Q7*) the engines show unexpected behavior. After registering a new query, C-SPARQL erroneously reports open windows. This behavior is only observed for sliding windows (slide is smaller than size of the window). $\text{SPARQL}_{\text{Stream}}$ produces incorrect results due to differences in the initial window timing and due to way aggregates are computed in the absence of matches. However, the authors note that the operational semantics of $\text{SPARQL}_{\text{Stream}}$ depend on the underlying stream processing engine. Hence, they are likely to change when the underlying engine is also changed, which in the end would lead to different results. CQELS has issues for queries *Q4*, *Q6*, and *Q7*. The root cause of these issues is that CQELS fails to remove RDF statements from the active window. Therefore, wrong aggregations and joins are computed leading to incorrect results.

3.2.2.4 Theoretical Benchmarking Approaches

Besides these experimental efforts, there is also a theoretical body of work on benchmarking RDF stream engines. In [DBDV13], which is a precursor to CSRBench, the authors

[17] Adapted from [DCB+13]

explain the need for functional testing of proposed engines. They argue that a precise evaluation and comparison of existing systems is complex. Moreover the authors state that SRBench and LSBench do not deal with issues related to window operators, because they assume that returned results are correct.

Scharrenbach et al. [SUM+13] state that most evaluations are conducted under incomparable and limited scenarios. To handle this problem they present three key performance indicators and seven commandments that help in defining stress tests for the evaluation of stream processing frameworks.

The key performance indicators are as follows:

- **Response time over all queries**
- **Maximum input throughput** (input data elements consumed per time unit)
- **Minimum time to accuracy and the minimum time to completion** for all queries (includes precision, recall, and error rate)

In order to obtain measurements of these Key Performance Indicators (KPIs) seven stress tests are defined, which are called the *seven commandments* for benchmarking semantic flow processing systems.

These stress tests are defined as follows:

- **Load Balancing**: detect bottlenecks under many operating queries and multiple processors.
- **Joins and Inference on Flow Data Only**: measure the performance of data joins on flow data.
- **Joins and Inference in Flow and Background Data**: measure the performance of data joining flow and background data.
- **Aggregates**: assess the performance of both shrinking and non-shrinking semantics.
- **Unexpected Data**: measure the ability to handle out-of-order data and data loss.
- **Schema**: evaluate an engine's handling of an increasing number and complexity of the statements in the schema.
- **Changes in Background-Data**: test how updates in background data affect performance.

To conclude, currently there are several benchmarks for semantic stream processing systems available. Table 3.3 compares them and shows that none of the benchmarks copes with all evaluation dimensions. A checkmark (✓) denotes *full support*, a hyphen (-) denotes *no support*, and a tilde (~) denotes *partial support* of the evaluation type.

Benchmark	Functional Evaluation	Correctness Evaluation	Performance Evaluation	Reviewed Engines
LSBench	✓	~	✓	C-SPARQL, CQELS, JTALIS
SRBench	✓	-	-	C-SPARQL, CQELS, $\text{SPARQL}_{\text{Stream}}$
CSRBench	-	✓	-	C-SPARQL, CQELS, $\text{SPARQL}_{\text{Stream}}$

Table 3.3: RSP benchmarks comparison

3.3 YABench framework

The YABench framework is organized into four parts: (i) a stream generator to create test data streams; (ii) supported engines to be benchmarked; (iii) an oracle to check correctness of results; and (iv) a reporting tool to generate visualizations based on experimental results. Because YABench operates in a streaming environment, we define the following requirements that need to be considered in this setting:

- *Valid, scalable, reproducible, and configurable input* (**R1**): Input data should be meaningful. This allows results to be more easily interpretable. Scalable and configurable input ensures running reproducible experiments under varying circumstances, for instance, high/low load and different window sizes.
- *Comprehensive correctness checking* (**R2**): It should be possible to check correctness of engines results, despite them using different operational semantics. Moreover we aim at measuring *real throughput*, i.e., how does input load affect correctness of results.
- *Flexible queries* (**R3**): Queries should be parametrizable, meaning that certain values can be changed based on test configurations. This allows to dynamically test engines based on same queries, but with varying configurations.
- *Useful reports* (**R4**): This is a minor requirement, but nonetheless important for practical reasons. Users should be able to define and run tests without changing the source code.

3.3.1 Architecture

YABench is designed around a modular architecture (cf. Figure 3.6) that decouples test configuration and execution. It allows to define tests in a declarative manner and can run complete benchmarking workflows with a single command.

The framework consists of four separately executable modules, i.e., the Stream Generator, RSP engine, the Oracle and the Runner which controls the overall execution flow of a test. The test configuration consists of a configuration file (*config.json*) and two query

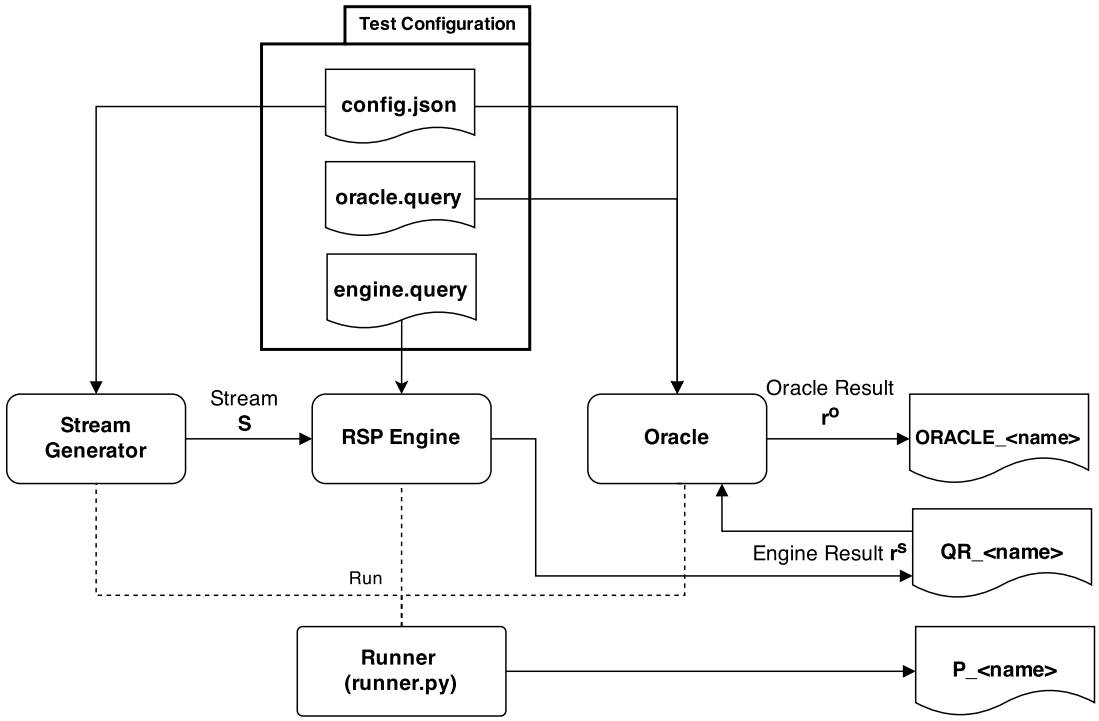


Figure 3.6: Architecture of the YABench framework

templates, *engine.query* for the engine and *oracle.query* for the oracle. The configuration file defines a set of tests that use the same query templates, but with varying parameters such as window size and slide.

The results of each test are the oracle results (*ORACLE_<name>*) and performance measurements (*P_<name>*) of the engine. These results can then be visualized by means of a provided reporting web application.

More details about the architecture and the test configuration can be found on the wiki of the project's GitHub repository[18].

3.3.2 Stream Generator

The Stream Generator satisfies *R1* and is used to create input data that is fed to the respective engines. It turned out to be more practical to decouple the steps of creating data, feeding it to the engines, and creating measurements from each other. The generator for the benchmarks in this work emulates an environmental monitoring scenario and draws on the LinkedSensorDataset[19] which is also used in SRBench [ZDCC12] and CSRBench [DCB+13]. The data set consists of weather observations from hurricanes in

[18] `http://github.com/YABench/yabench` (accessed 16 June 2016)
[19] `http://wiki.knoesis.org/index.php/SSW_Datasets` (accessed 16 June 2016)

the USA. We selected this simple data model for two reasons: (i) it makes our work comparable to previous work, particularly to CSRBench (cf. Section 3.4); (ii) having such a simple and generic model allows for scenario parameterization, e.g., by changing the *number of simulated weather stations* to vary *load on an engine*. More complex data flows would make it more difficult to discriminate effects such as which parameter influences which measurable performance indicator of an engine. To simulate more complex data flows, YABench can easily be extended with additional Stream Generators.

Figure 3.7 illustrates the structure of the data model. A central element is `weather:TemperatureObservation`, which represents a single observation. This observation is connected to actual measure data via `om-owl:result`. The `om-owl:observedProperty` indicates which environmental condition was sensed by the `ssw:system`. The system represents a sensor which is creating measurements. It is connected to the observation via the `om-owl:procedure` relation.

The process generates an RDF stream $\mathbb{S}$ based on an input function $gen(s, i, d, r, n)$, where s denotes the number of simulated systems, i denotes the time interval between two measurements of a single station, d denotes the duration of the generated output stream, r denotes a seed for randomization to vary the timestamps of initial measurements of each system, and n defines the generator which should be used. Currently, one such stream generator is implemented in YABench, which uses the data model and workflow outlined above. Input load for experiments can be varied with the s and i parameters. The combination of parameters s, i, and r ensures reproducibility, because they guarantee that the exact same stream is generated every time for a given parameter set.

Listing 3.1 shows an excerpt of example data generated by the stream generator. The

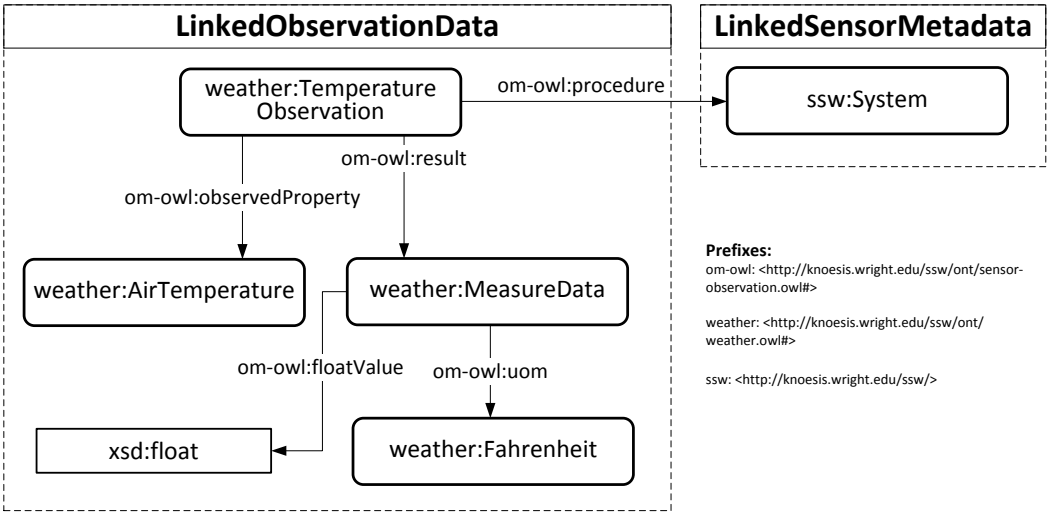


Figure 3.7: Data model of generated streams based on LinkedSensorData.

```
ssw:Observation_AirTemperature_A521_0 a weather:TemperatureObservation "10" .
ssw:Observation_AirTemperature_A521_0 om-owl:observedProperty weather:_AirTemperature "10" .
ssw:Observation_AirTemperature_A521_0 om-owl:procedure ssw:System_A521 "10" .
ssw:Observation_AirTemperature_A521_0 om-owl:result ssw:MeasureData_AirTemperature_A521_0 "10" .
ssw:MeasureData_AirTemperature_A521_0 a om-owl:MeasureData "10" .
ssw:MeasureData_AirTemperature_A521_0 om-owl:floatValue 95.0^^xsd:float "10" .
ssw:MeasureData_AirTemperature_A521_0 om-owl:uom weather:fahrenheit "10" .
```

Listing 3.1: Example output of one sensor measurement.

data is persisted in a file using the N-Triples format[20] (for the sake of clarity we use prefixes) which we extend with a timestamp t that is equal to the number of milliseconds from the beginning of the stream. The timestamp will be used when data is streamed to the engines.

3.3.3 Engines

We use YABench to evaluate two engines, i.e., C-SPARQL 0.9.5[21], and CQELS 1.0.0[22]. After a stream has been generated, the YABench's engine component invokes a function $stream(d, q, s)$, where d denotes the destination of output files, q defines the continuous query which will be registered at the engine, and s defines the input stream, which was previously generated. At this stage output files will be created for performance measurements and query results.

YABench wraps each engine to allow stream data feeding under controlled circumstances. The input RDF stream $\mathbb{S}$ consists of a sequence of timestamped triples in non-decreasing time order in the following form:

$$\mathbb{S} = ((\langle s, p, o), t_0\rangle), (\langle s, p, o), t_1\rangle), ...)$$

The wrapper iterates over the input stream $\mathbb{S}$ and feeds sets of RDF statements with same timestamps to the engine, hence,

$$F_{t_0} = \{(\langle s, p, o)\rangle \mid (\langle s, p, o), t_0\rangle \in \mathbb{S}\ \}$$

while respecting time intervals $i = t_1 - t_0$ between feeds F_{t_0} and F_{t_1}.

While feeding the engines with the graphs, we continuously take performance measurements, i.e., absolute and relative memory consumption, CPU usage, and the number of threads spawned.

3.3.4 Oracle

To check the correctness of query results and thereby satisfy *R2*, we implemented an oracle. The implementation is inspired by the oracle used in CSRBench, but built on top of Jena ARQ[23] and extended with means to measure more granular metrics, which are

[20] `http://www.w3.org/2001/sw/RDFCore/ntriples/` (accessed 16 June 2016)
[21] `http://github.com/streamreasoning/CSPARQL-engine` (accessed 16 June 2016)
[22] `https://code.google.com/p/cqels/` (accessed 16 June 2016)
[23] `https://jena.apache.org/documentation/query/` (accessed 16 June 2016)

computed per window: precision and recall, delay of query results, number of triples in the window, and number of tuples in query results.

The semantics used by the oracle can be configured by means of configuration files. This means that the specification of the oracle can be changed as desired, that is, we change it according to which engine we want to test and, therefore, emulate. Report policy parameters are provided to emulate either CQELS (*OnContentChange*) or C-SPARQL (*OnWindowClose*). Hence, we are able to provide correct results that account for the respective report policy.

The oracle checks the results (recorded in *QR_<name>* file) of a continuous query q by using the same input stream $\mathbb{S}$ and an equivalent, but static, SPARQL query q' (*oracle.query*). It takes into account the report policy [BDD+10] which the given engine applies as well as window size (α) and slide (β) parameters of the continuous query q.

The following report policies are supported by the oracle:

- *Content change*: reporting is done only when the content of the active window changed. Supported by CQELS.
- *Window close*: reporting is done only when the active window closes. Supported by C-SPARQL.

To check correctness, the oracle performs the following steps:

(i) Determine the *scope* $[t_s, t_e)$ of the next window that will report based on the given window size α, window slide β, and the required report policy.

(ii) Use the *scope* $[t_s, t_e)$ to select window *content* from the input stream $\mathbb{S}$ where the relevant triples are defined as

$$F_{t_s,t_e} = \{(\langle s,p,o\rangle) \mid (\langle s,p,o\rangle, t\rangle \in \mathbb{S}, t_s \leq t < t_e\}$$

(iii) Compute the expected result by executing the SPARQL query q' on F_{t_s,t_e} on the query engine.

(iv) Compare the result of this query with the next result of continuous query q and compoute precision/recall metrics.

(v) Compute the remaining metrics, i.e., delay, window size, and result size.

3.3.4.1 Delay

The delay d with which the engine outputs query results is computed as the difference between the end timestamp of the oracle window $t_{e_o}^{\mathbb{W}_i}$ and the timestamp when the engine outputs the result for this window $t_{e_s}^{\mathbb{W}_i}$. The timestamp when the result is recorded is stored in *QR_<name>* file.

3.3.4.2 Gracious mode

In addition to the algorithm described above, the oracle implements a new method which takes known issues of supported report policies into account and aims to eliminate them.

- *Content change*: lower precision/recall may be observed because of delayed purging of content of the previous window from the engine's active window;
- *Window close*: lower precision/recall may be observed because of the shift of the engine's active window scope forward on the timeline which can be caused by high load, for instance.

In case of *content change* policy, the oracle aims to compensate for low precision/recall by performing the following steps:

(i) Determine the *scope* $[t'_s, t_e)$ of the next window that will report. The left border of the *scope* is $t'_s = t_s - \beta$.

(ii) Repeat steps two to four of the algorithm above.

(iii) Move t'_s to the time (t) of the next triple which is $t > t'_s$ until recall becomes lower than with the previous t'_s and sets t'_s to the time with the highest recall.

In case of the *window close* policy, the oracle does the following:

(i) Move t_s of the *scope* $[t_s, t_e)$ to the time (t) of the next triple which is $t > t_s$ until recall becomes lower than with the previous t_s, and sets t_s to the time with the highest recall.

(ii) Move t_e of the *scope* $[t_s, t_e)$ to the time (t) of the next triple which is $t > t_e$ until precision becomes lower than with the previous t_e, and sets t_s to the time with the highest precision.

3.4 Validation against CSRBench

The validation against CSRBench ensures that YABench produces equivalent results. This validates the correct operation of YABench's internals, however, in later experiments (cf. Section 3.5) we show that our framework allows to perform evaluations that go beyond the scope of CSRBench, i.e., providing more granular metrics. The source code and instructions on how to run the validation are published on GitHub[24].

The methodology for the validation is as follows:

[24] `https://github.com/YABench/csrbench-validation` (accessed 16 June 2016)

- We convert the original data used by CSRBench to the N-Triples format, and extend it with timestamps to emulate an input stream load of one observation per second. By doing so, the data stream is the same as the output of our stream generator.
- For each of the seven CSRBench queries we need to setup tests configurations. These configuration files define parameters such as window size, window slide, and eventual filter parameters, which are subsequently needed the queries are registered at the engine.
- For each engine and for each of the seven CSRBench queries we then execute the test, i.e., feeding the engines with the input stream and registering queries with the defined parameters.
- After all tests are finished, we compare the results of CSRBench which can be found at GitHub[25] with the results created by YABench.

Table 3.4 summarizes the results of the validation. Results were compared and inspected manually. Checkmarks indicate that YABench produced results equivalent to CSRBench. Columns of CSRBench which contain an × denote that the respective engine did not produce correct results. In all such cases, YABench also indicated that the engine did not deliver correct results.

Checkmarks denoted with one asterisk (*) indicate that YABench produced mostly identical results, but that some results were missing. This occurred in some cases were triples were close to a window border. In these cases, we found that in C-SPARQL such triples may fall either into the scope of $\mathbb{W}_n$ or $\mathbb{W}_{n+1}$. This can be attributed to timing discrepancies, which we encountered after running benchmarks multiple times and/or on different systems. However, we verified that all results are present, if not in the correct window, then at latest in the subsequent window.

Crosses in the YABench columns indicate that results did not match those of CSRBench experiments. This is expected behavior, however, since the same queries did not pass correctness tests of CSRBench in the original tests either. The cross denoted with two asterisks (**) indicates that the query ($Q7$) did not run on the CQELS engine.[26]

We can conclude that, besides minor, well-explained inconsistencies, YABench reproduces the results of CSRBench. Beyond the scope of previous benchmarks, however, YABench employs a more comprehensive approach that allows (i) to define experiments including test data, input load parameters, and queries, (ii) to perform experiments that consider the varying operational semantics of the tested engines, and (iii) to conduct in-depth analyses based on new throughput, delay, and correctness metrics.

These capabilities are used for experiments discussed in the following section.

[25] `https://github.com/dellaglio/csrbench-oracle-engines` (accessed 16 June 2016)
[26] The system crashed before returning the query results.

Query	C-SPARQL		CQELS	
	CSRBench	YABench	CSRBench	YABench
Q1	✓	✓	✓	✓
Q2	✓	✓	✓	✓
Q3	✓	✓*	✓	✓
Q4	✓	✓	×	×
Q5	×	×	✓	✓
Q6	✓	✓*	×	×
Q7	✓	✓*	×	×**

Table 3.4: Results of YABench validation against CSRBench results[27]

3.5 Experimental setup

We use YABench and its oracle to perform experiments with two engines, C-SPARQL and CQELS. In particular, we are interested in how the correctness of results is affected by changes in the input data streams. Whereas previous correctness metrics exclusively focused on checking whether engine results are included in the oracle results, i.e., a yes/no evaluation, we provide more granular metrics. We use precision and recall calculations in combination with performance metrics which are computed for each window. These metrics uncover issues that can be caused by, for instance, shifting of window borders under load, leading to lower precision or recall values. We measure an engine's delay in delivering results and the amount of RDF statements inside a window's scope to understand and be able to explain low retrieval rates. YABench is the first RSP benchmark to provide a comprehensive picture of an engine's behavior under stress.

We reuse the queries introduced by CSRBench, however, for each query we parametrize window size α, window slide β, and filter values f (*R3*). For the input streams, we control the number of stations s to simulate low, medium, and high load scenarios. The interval time i between measurements will be 1s and the duration of each experiment is 30s.

Performance measurements, such as memory consumption, are taken at regular intervals, i.e., 500ms. Because all other metrics observe characteristics of the windows, they are taken and displayed on a per window basis. We replicated each experiment ten times and illustrate the distribution of result metrics obtained for precision, recall, and delay as boxplots. For performance indicators such as memory consumption, YABench-reports generates aggregated (averaged) line charts. Where appropriate, we will also discuss detailed observations for individual runs.

[27] Explanation for cells which include asterisks are given in the text.

We use the CSRBench queries available on the W3C wiki[28] for our experiments, which were executed on an Intel Core i7-3630QM @ 2.4GHz, Quad Core, 64bit, 12 GB RAM running Windows 7 Professional. For the sake of completeness we include the (slightly adapted[29]) queries in Appendices A–C of this thesis. Complete results are published on GitHub[30] and can be visualized with our web application *YABench-reports*.

3.5.1 Experiment 1 (*Q1*): Select

This experiment uses a simple `SELECT` statement combined with a `FILTER` asking for the latest temperature observations above a specified threshold and the sensor which took the measurement (cf. Appendix A). We run the experiment with the following parameters: $\alpha = 5s$, $\beta = 5s$, $s = 50/1000/10000$ (small/medium/big), $i = 1s$.

3.5.2 Experiment 2 (*Q4*): Average

The second query makes use of the aggregation function `AVG` combined with a `FILTER` to return the average temperature value over a given window (cf. Appendix B). To answer such aggregate queries, depending on the report and tick policy as well as window content and window size, stream processors typically face high resource costs. Because CQELS does not comply with the semantics of `AVG` as defined by SPARQL 1.1[31], we had to implement a custom `AVG` operator that returns an empty result if there are no matches. We run the experiment with the following parameters: $\alpha = 5s$, $\beta = 5s$, $s = 1/1000/10000$, $i = 1s$.

3.5.3 Experiment 3 (*Q6*): Large change

This query returns sensors that made two observations (of different timestamps) with a variation between measurements higher than a given threshold (cf. Appendix C). The query uses the `SELECT` keyword to ask for shifts of measured values over time from the same sensor. To execute this query, engines must be able to join triples over different timestamps. In order to produce meaningful and comparable results for both engines in this experiment, we slightly changed the number of simulated stations (s) and ran the experiment with the following parameters: $\alpha = 5s$, $\beta = 5s$, $s = 50/200/500$, $i = 1s$.

3.5.4 Experiment 4 (*Q6*): Large change gracious

This experiment is designed to reveal issues of lower precision/recall values which we observed during experiment 3 for both engines. For CQELS the reason for decreasing precision/recall is delayed deletion of window content, for C-SPARQL slight window shifts are responsible. This led us to the development of a so called *gracious* mode where the

[28] `http://www.w3.org/wiki/CSRBench` (accessed 16 June 2016)

[29] Variables which are substituted with actual values based on the configuration file are denoted with a dollar ($) sign in the queries.

[30] `http://github.com/YABench/yabench-one` (accessed 16 June 2016)

[31] `http://www.w3.org/TR/sparql11-query/#defn_aggAvg` (accessed 16 June 2016)

oracle eliminates these issues resulting in both high precision/recall and the possibility to detect new issues unrelated to potential window delays. Hence, this experiment shows the effects of the *gracious* mode by comparing its results with the *non-gracious* mode as well as discussing and explaining the differences. In *non-gracious* mode the oracle does not account for any issues and expects ideal behavior of the engines.

We ran two similar tests for both engines with the following parameters, one of them in *gracious* mode and the other one in *non-gracious* mode: $\alpha = 5s$, $\beta = 5s$, $s = 1$, $i = 1s$.

3.6 Discussion

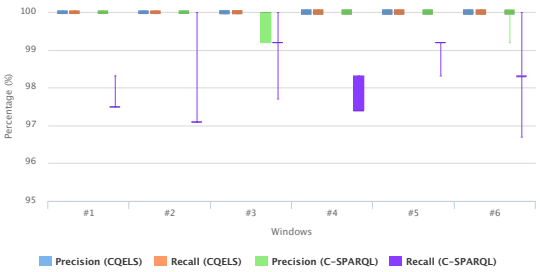
The YABench framework provides a reporting web application (*YABench-reports*). Based on results of the oracle and performance measurements it displays three graphs, i.e., (i) a precision/recall graph that includes indicators for the windows, (ii) a graph showing delay of result delivery, and expected/actual result size, and (iii) a graph providing performance measurements.

3.6.1 Experiment 1: Select

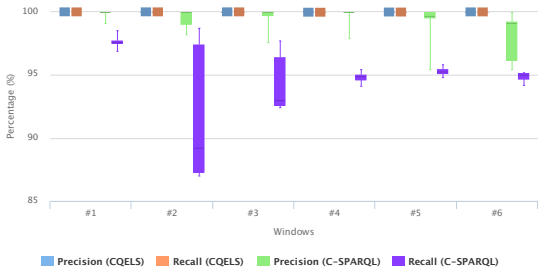
Figures 3.8a–c illustrate the results of the first experiment; they show boxplots of precision and recall values for each of the three load scenarios (small: $s = 50$, medium: $s = 1000$, and big: $s = 10000$); Figures 3.9a–c show boxplots of the observed delay; and Figures 3.11a–c show line charts of an engine's memory consumption during the stream feeding and query evaluation.

We found that CQELS maintains 100% precision and accuracy under low load, whereas C-SPARQL achieves slightly lower values (precision is at 100% except for window three and four, the mean for recall ranges between 97% and 100%). Generally, we observe that recall is lower than precision for C-SPARQL. The shifting of the actual engine windows compared to the ideal expected windows from the oracle due to delays is responsible for this behavior which will be explained in more detail later (cf. Figure 3.12).

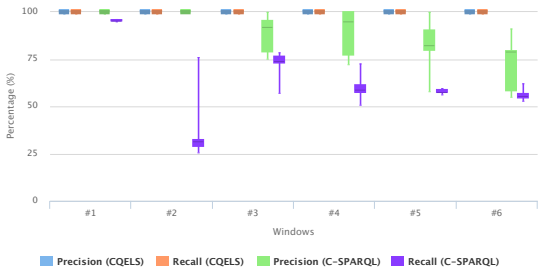
We observed similar behavior under medium load (cf. Figure 3.8b). For this simple query, CQELS still scores perfect precision and recall, whereas we observe deteriorating effects for C-SPARQL. The recall values from window two and three have a particularly high spread. The mean of all recall measurements lies between 89% and 97%, which is very high. The spread can be explained by the higher delays of the first two windows (cf. Figure 3.9b). Because C-SPARQL delivers results upon the closing of a window, the delay has an effect on precision and recall. This is not the case in CQELS, where delay in result delivery does not necessarily mean that the window content — and consequently the computed results — is incorrect. In fact, for CQELS the opposite is the case, meaning that delayed results still provide correct results. This is also taken into account by our oracle. The oracle is aware that results can arrive with a delay and still applies its precision/recall measurements to these results.



(a) Results of *Experiment 1*, precision/recall, $s = 50$



(b) Results of *Experiment 1*, precision/recall, $s = 1000$



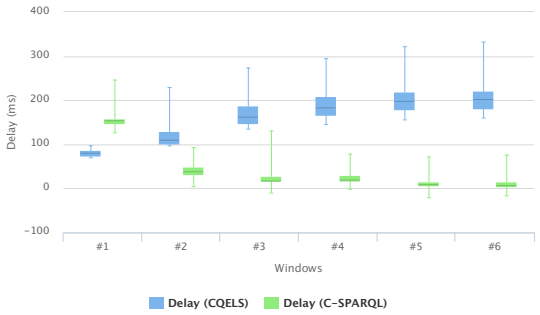
(c) Results of *Experiment 1*, precision/recall, $s = 10000$

Figure 3.8: *Experiment 1*, precision and recall

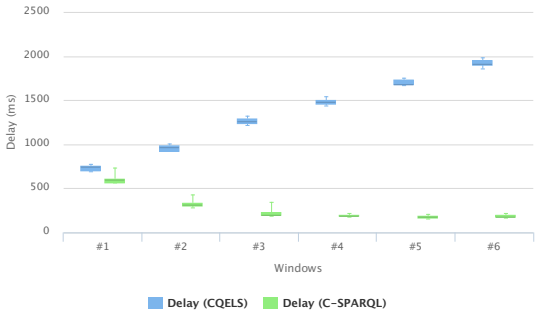
Results under high load are shown in Figure 3.8c. The results are similar to the low and medium load scenarios. For C-SPARQL, the first window yields very good recall and even better precision. Again, recall of the second window has a high spread ranging from 25% to 76%. The remaining windows show recall with a mean varying between 55% and 74%. Considering the high throughput in this scenario (one window of 5s contains about 24.000 result triples out of 350.098 triples), precision (between 78% and 100%) is still high.

Delay of result delivery under low load is depicted in Figure 3.9a. For CQELS mean result delivery varies between 81.5ms (window one) and 201.5ms (window six). The values rise steadily, but even out for the last three windows. Interestingly, delay in C-SPARQL exhibits the opposite characteristics. The first window always yields longer delay (mean = 153.5ms), whereas the following windows show short delays between 7ms and 38ms. Delay can also be negative, when results are delivered too early. This is the case if for a window size of 5s, where results are delivered before 5s have passed.

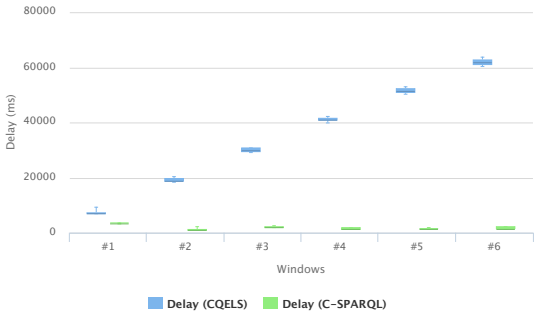
Figure 3.9b shows the delay for medium load. We observe similar behavior as before, but to a greater extent. CQELS delays do not even out anymore for the last windows, but continue to increase. Except for the first window, which again shows higher delay (mean = 594ms), C-SPARQL's mean delays vary between 176.5ms and 313ms.



(a) Results of *Experiment 1*, delay, $s = 50$



(b) Results of *Experiment 1*, delay, $s = 1000$



(c) Results of *Experiment 1*, delay, $s = 10000$

Figure 3.9: *Experiment 1*, delay

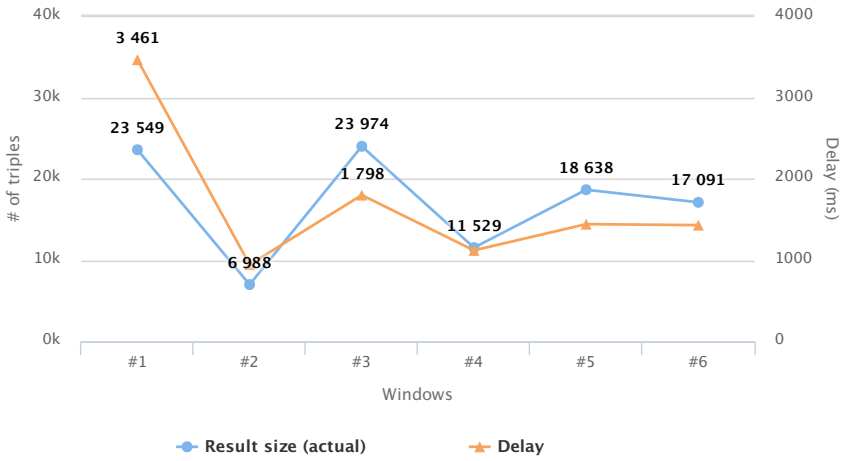


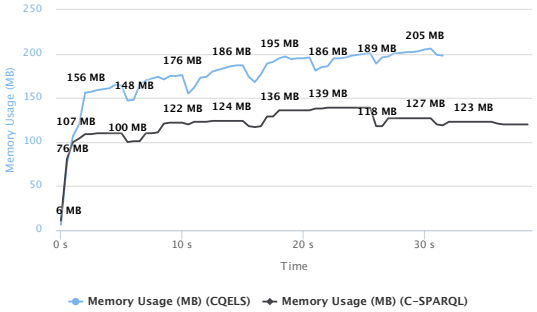
Figure 3.10: Correlation between result size and delay for C-SPARQL

For the high load scenario we can see in Figure 3.9c that, in contrast to the lower load results, the second window of C-SPARQL yields lower delay than the consecutive windows. However, again, the mean delays from the second window on are constantly between

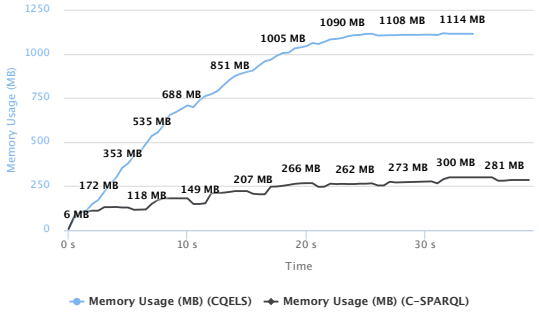
1000ms and 2000ms, which is a broad range as can be expected for an engine under high pressure. The delay results for CQELS behave similar to the results of the medium load scenario. Delay increases from 8000ms (first window) up to 60000ms (sixth window).

Finally, we found that YABench reveals a correlation between the result size and delay times, as shown in Figure 3.10, which shows both metrics for a single test run. We see that delay times increase when result size increases. This relation is expected, but YABench allows to quantify the influence of large amount of result bindings on an engine's performance.

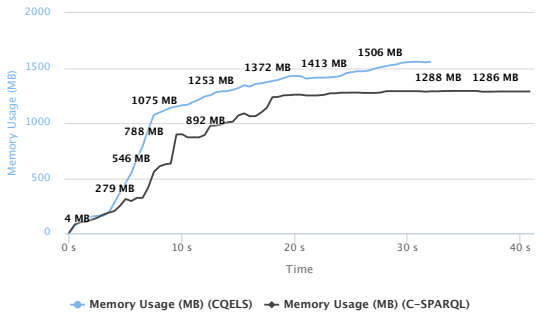
Figures 3.11a–c provide details about the performance of both engines. Due to the different reporting policies, the C-SPARQL experiments last longer than the CQELS experiments. This is reflected on the y-axis of the graphs. C-SPARQL is more memory efficient and exhibits a moderate increase in memory consumption between low (mean = 123MB) and medium (mean = 250MB) load. For both engines, the removal of window content is apparent in the charts — particularly in Figure 3.11a — in the form of rapid decreases after every five seconds, i.e., the defined window size. Under medium load, memory consumption of CQELS rises to about 1100MB where it then flattens out. Both engines show similar behavior under high load (cf. Figure 3.11c), where the charts show a



(a) Results of *Experiment 1*, memory consumption, $s = 50$



(b) Results of *Experiment 1*, memory consumption, $s = 1000$



(c) Results of *Experiment 1*, memory consumption, $s = 10000$

Figure 3.11: *Experiment 1*, memory consumption

steep increase in memory consumption until ten seconds. Beyond that, the graph flattens out again with a maximum of 1506 MB (CQELS) and 1288 MB (C-SPARQL).

For C-SPARQL precision always is above recall, when both values decrease. Because higher load leads to bigger delays in query result delivery, precision and recall decrease.

The observed delay supports the conclusion that the actual windows are shifted, therefore, deviating from the ideal windows computed by the oracle as is shown in Figure 3.12. Given a query which asks for all statements occurring on stream $\mathbb{S}$, a delay between start and end timestamps of the expected window computed by the oracle $\mathbb{W}_e$ and the actual window $\mathbb{W}_a$ by the engine, can be observed. This is also the reason for lower precision and recall values. $\mathbb{W}_e$ contains only one (s_2) of three relevant statements (blue filling), hence, recall $r = 1/3$. Out of the two selected statements of $\mathbb{W}_e$ ($\{s_1, s_2\}$) only the latter one is relevant, hence, precision $p = 1/2$. In other words, whereas the scope of an ideal window is $[t_s, t_e)$, the scope of shifted windows adds a delay to the start and end timestamps and is denoted as $[t_s + d_s, t_e + d_e)$. The delays d_s and d_e can be different due to timing issues of engines. This explains different decrease in precision and recall.

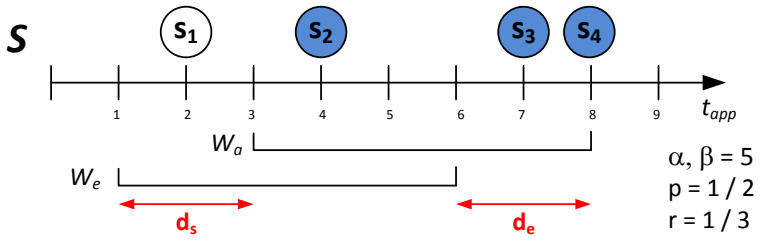
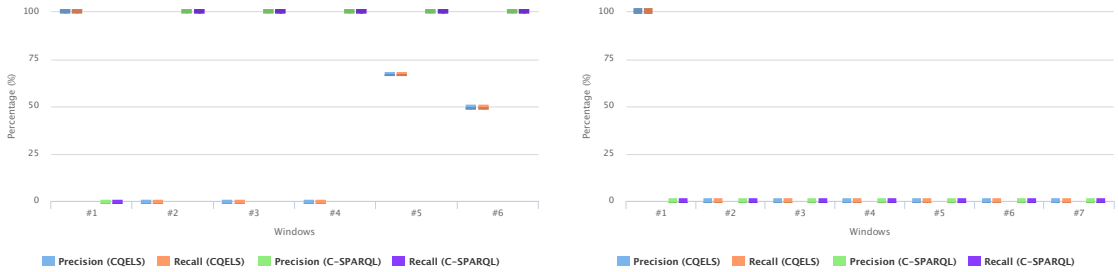


Figure 3.12: Lower precision and recall due to delay of actual window $\mathbb{W}_a$

3.6.2 Experiment 2: Average

Results of experiment two are shown in Figures 3.13a–b. This experiment evaluates the capability to deal with an aggregate query averaging the temperature observations over a window. It is important to note that YABench only matches result triples successfully if the output of an engine is exactly the expected average value. In other words, if an engine outputs *95* as a result of this query, but the expected average should be *96*, then precision and recall will be zero. This implies that a single missed triple (due to, e.g., window delays in the case of C-SPARQL) is sufficient to produce low precision and recall values. For CQELS we discovered the reason for low precision and recall. The engine delays the deletion of window content, resulting in incorrect average calculation. This issue is explained in more detail in experiment four.

As a result of these observations, we adapted the low load scenario to only simulate a single station. This reduces missing triples due to window delays and, hence, we expect higher precision and recall. Figure 3.13a shows the precision/recall measurements for low load where the number of stations s equals one. CQELS yields 100% precision and



(a) Results of *Experiment 2*, precision/recall, $s = 1$

(b) Results of *Experiment 2*, precision/recall, $s = 1000$

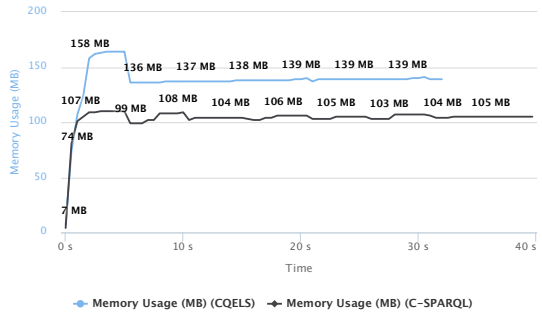
Figure 3.13: *Experiment 2*, precision and recall

recall only in the first window. This confirms our assumption of issues in deletion of window content: At the first window, there is no possibility to forget to delete outdated window content, simply because there is no outdated content, yet. In the consecutive windows, precision and recall drops to zero. This is where content becomes outdated, but obviously not properly deleted by CQELS, resulting in lower values.

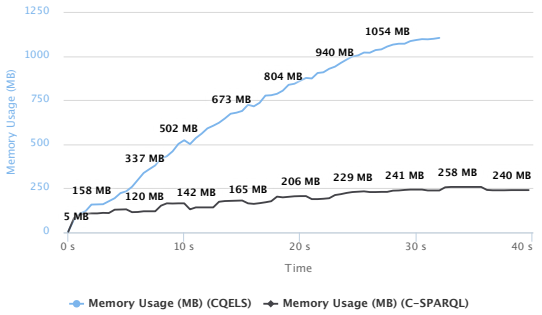
Interestingly, for C-SPARQL we observe the opposite behavior. In the first window precision and recall are at zero. The consecutive windows, however, show perfect precision and recall when the average temperature over the measurements of a single station is calculated. Upon manual inspection we found that C-SPARQL ignores the first streamed triple for average calculation, resulting in 0% for precision and recall of the first window, which appears to be a bug in the implementation. For the medium load scenario, Figure 3.13b shows that — as expected and already explained — both engines show low precision and recall, except CQELS for the first window. The values do not increase under high load which is why we omit to show the graphs.

Figures 3.14a–c show memory consumption for all three scenarios. Generally, both engines show similar behavior as in the first experiment. Memory consumption of C-SPARQL is lower in experiment two than in experiment one. Moreover, the values increase a bit slower under high load than in the first experiment.

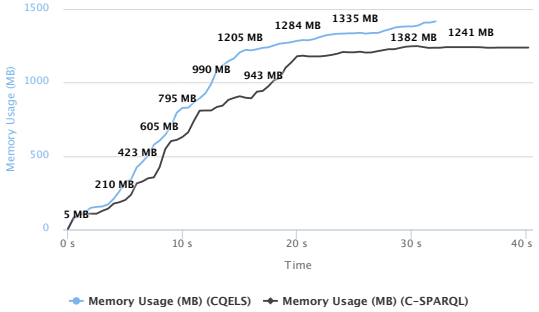
This experiment revealed additional characteristics of the tested engines. For medium and high load, CQELS shows increased delays in result delivery for the average query used in this experiment. Even though each experiment run lasts 30 seconds, under medium load the last result was delivered between 7min 8sec (min) and 7min 21sec (max) for our ten test runs. Under high load the last result, which should be delivered after 30 seconds, actually appeared between 3h 25min (min) and 3h 45min (max) late. These delays make it impractical for our oracle to perform any measurements regarding retrieval performance in this experiment. We did not observe the same delays in the first experiment where the query only requires a simple graph pattern match. We can therefore conclude that the excessive delay is due to the complexity added through the aggregate query. Due to its report strategy, CQELS has to recompute and emit the aggregate for each new arriving



(a) Results of *Experiment 2*, memory consumption, $s = 1$



(b) Results of *Experiment 2*, memory consumption, $s = 1000$



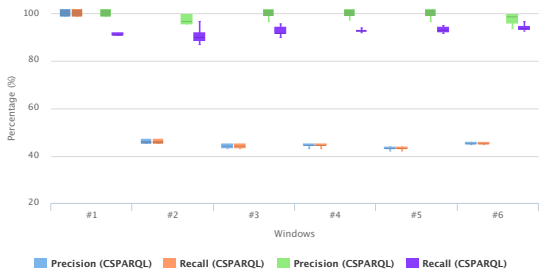
(c) Results of *Experiment 2*, memory consumption, $s = 10000$

Figure 3.14: *Experiment 2*, memory consumption

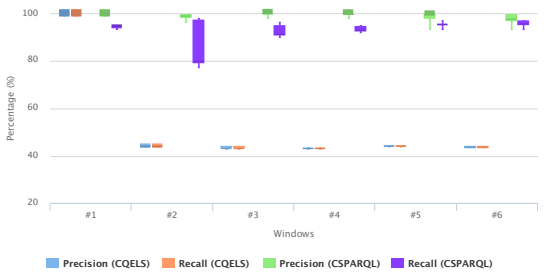
triple. Moreover, calculating aggregations is more complex than checking for a simple pattern match. The engine has to fetch all relevant statements in the window, then do the computation, and finally output the result. Different types of optimizations could be applied here to improve aggregate calculation while avoiding the need to re-fetch the whole window content after each new streamed triple. In case of an average aggregate, a *moving average* could be used. From a practical perspective, it may not be useful to receive a new aggregate value for each new arriving triple. Consequently, the explained behavior may not pose a disadvantage in real use cases, as it frequently is not required to receive new aggregates in such high frequency as tested here. However, it would be useful to be able to control when an engine produces new output, i.e., the *report strategy* [BDD+10].

3.6.3 Experiment 3: Large change

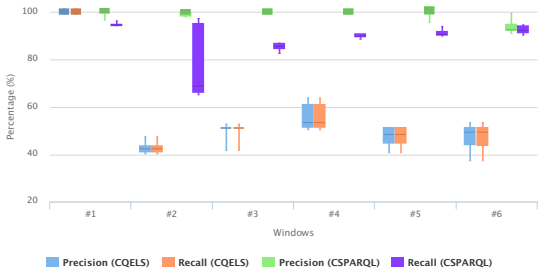
In Experiment three we use YABench to evaluate an engine's capability to process `SELECT` queries joining triples of different time stamps combined with a `FILTER` clause. Figures 3.15a–c show precision and recall for each window. For CQELS, the graphs show that the engines fails to delete outdated window content: For the first window precision



(a) Results of *Experiment 3*, precision/recall, $s = 50$



(b) Results of *Experiment 3*, precision/recall, $s = 200$



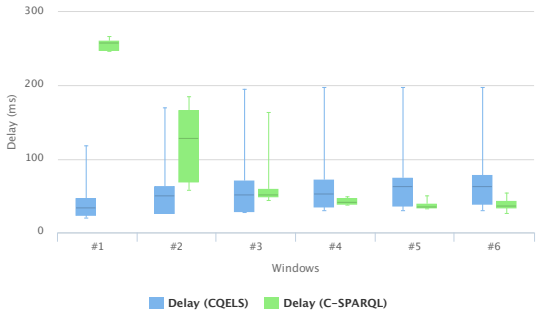
(c) Results of *Experiment 3*, precision/recall, $s = 500$

Figure 3.15: *Experiment 3*, precision and recall

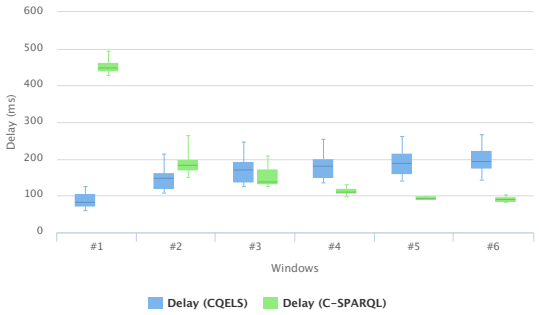
and recall are at 100% while for the consecutive windows they constantly lie between 40% and 60% due to incorrect window content. Counter-intuitively, under high load precision and recall is higher, but fluctuates more than under low and medium load. C-SPARQL handles this type of query well. With only two exceptions (recall in the second window under medium and high load) precision and recall are always above 80%. Similar to the other experiments, recall is a bit lower than precision. The recall of the second windows show particularly high fluctuation due to higher delays observed in the first windows.

Figures 3.16a–c show the result delivery delays. Under low load we observe slightly increasing delay for CQELS, however, it becomes stable at about 60ms. For C-SPARQL we observe high delay in the first window which consequently becomes less in the second, third, and remaining windows, evening out little below CQELS at 37ms (mean of window four). Similar behavior, but to a greater extent, is shown under medium load. Delay of CQELS becomes stable at 200ms, whereas C-SPARQL remains at 100ms for the second half of the experiment. Under high load CQELS is not able to catch up with the incoming data stream anymore, since delay keeps increasing over the course of the experiment peaking at twelve seconds. In contrast, C-SPARQL performs well with delays of 200ms.

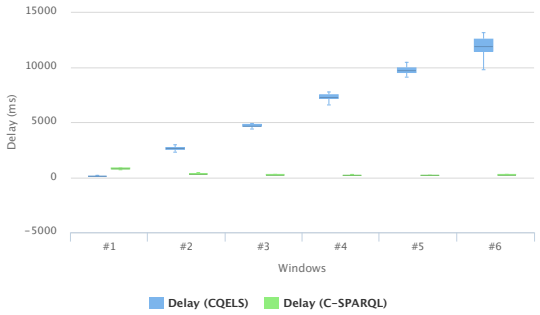
Memory consumption graphs are shown in Figures 3.16d–f. CQELS's need for more memory increases quicker than for C-SPARQL. In general, C-SPARQL's memory consumption does not increase over time. Compared to low load memory consumption of experiment one (cf. Figure 3.11a), where CQELS requires 200MB of memory, we observe an increase in memory demand here (250MB), which can be explained by the higher



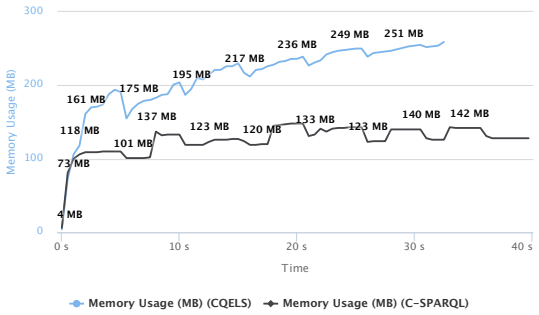
(a) Results of *Experiment 3*, delay, $s = 50$



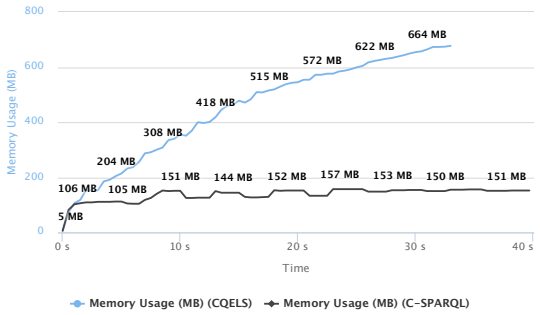
(b) Results of *Experiment 3*, delay, $s = 200$



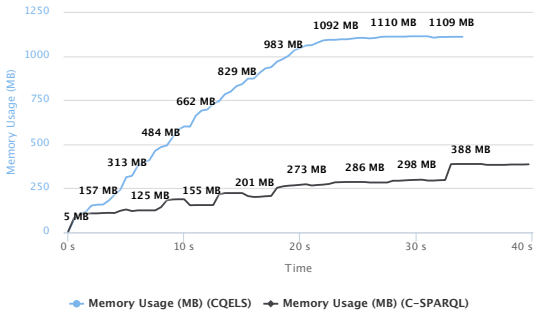
(c) Results of *Experiment 3*, delay, $s = 500$



(d) Results of *Experiment 3*, memory consumption, $s = 50$



(e) Results of *Experiment 3*, memory consumption, $s = 200$



(f) Results of *Experiment 3*, memory consumption, $s = 500$

Figure 3.16: *Experiment 3*, delay and memory consumption

query complexity in this experiment.

To sum up, experiment three reveals good precision and recall performance for CQELS, if we ignore the mentioned window purging issue. For the high load scenario, however, delay times start to increase up to a point that makes stream processing impractical. Memory consumption increases quickly, similar to previous experiments. C-SPARQL shows good performance and constantly low delay. Only recall for the second windows

fluctuates, which can be explained by the high delay in the first windows.

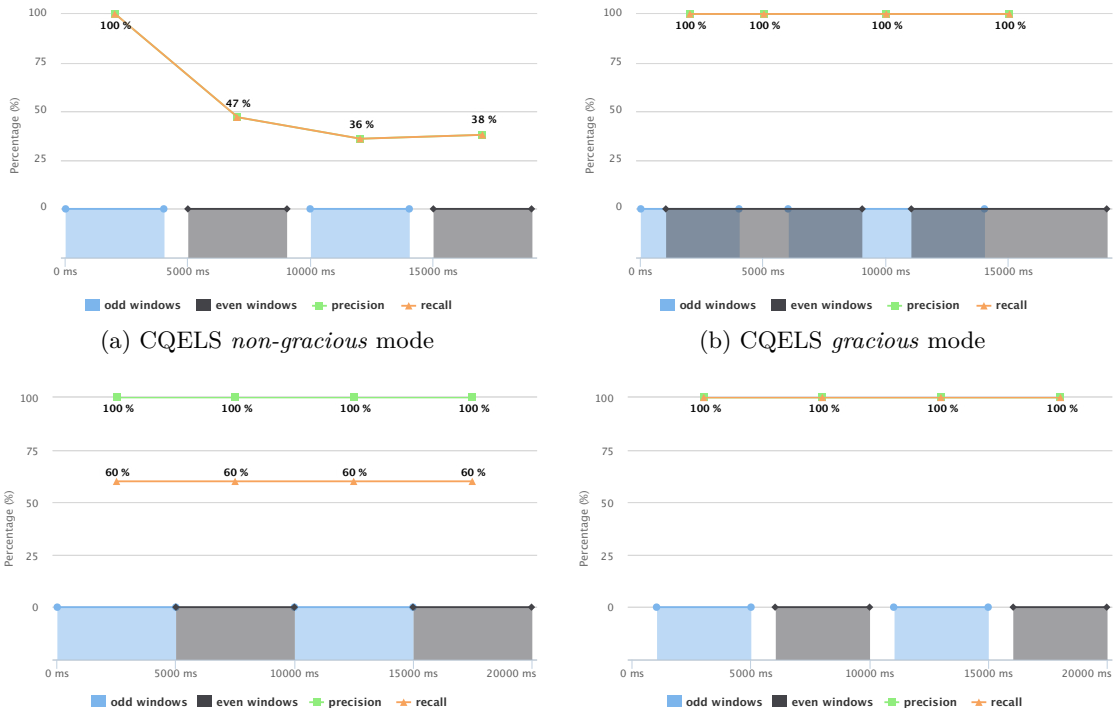
3.6.4 Experiment 4: Large change gracious

Experiment four investigates and explains issues that we experienced while manually testing the engines. Under certain conditions, which are emulated in this experiment by using the same query as in experiment three, we observed that precision and recall values are decreasing. For CQELS, which employs a *content change* report strategy, these lower values are caused by delayed purging of active window content. With purging we mean deletion of elements from the content of a window. As time in a streaming setting moves on, elements exit the scope of windows, hence, engines are responsible of correctly maintaining the content of windows. In C-SPARQL, which employs a *window close* report strategy, the values can be explained by the shifting of an engine's active window forward on the timeline (cf. Figure 3.12).

To investigate the root causes of these issues we implemented a *gracious* mode. In this mode, the oracle adjusts its window scope to match the window scope of an actual window, even though the actual window may contain incorrect elements (cf. Section 3.3.4.2). This has two consequences: First, precision/recall values increase, because *gracious* mode reverts the effects of incorrect window content. This allows us to confirm our assumptions on why low precision/recall values were observed. Second, we are able to visualize window borders which were actually used internally by the engines. By doing so, we unveil differences between expected and actual windows. Expected windows are windows which we would expect from a correctly implemented engine with zero delays.

Figure 3.17 shows the oracle results of the tests for CQELS and C-SPARQL engines. As we can see in Figure 3.17a, precision and recall values decrease after the first window for CQELS in *non-gracious* mode. This confirms the issues we experienced during manual testing. Figure 3.17b shows results of the same test, but with *gracious mode* enabled. Here we can see that precision and recall values are both at 100%. YABench does so by shifting the window borders of the oracle until reaching maximum precision and recall values. As a result we can see the adapted window borders at the bottom of the charts. In Figure 3.17a we observe that the oracle had to shift the window starts to the left in order to reach high precision and recall. This indicates that the engine forgets to delete outdated elements from the content of the active window for the query which is used in this experiment.

Similar, although less pronounced, effects can be observed for C-SPARQL. In *non-gracious* (cf. Figure 3.17c) mode we see low recall values. The reason is that due to slight delays in result delivery, not all relevant items appear in the window content of the oracle. Shifting window borders again leads to precision and recall of 100%, as shown in Figure 3.17d. In contrast to CQELS, here we need to shift the window start to the right in order to improve the results. This is visualized again at the bottom of the figure. This implies that the actual window of the engine was slightly delayed.



(a) CQELS *non-gracious* mode

(b) CQELS *gracious* mode

(c) C-SPARQL *non-gracious* mode

(d) C-SPARQL *gracious* mode

Figure 3.17: *Experiment 4* precision/recall results in *gracious* and *non-gracious* modes

Finally, Table 3.5 presents observations which we made during the experiments on CQELS. It shows the retrieval time of the final result of the tests for small (S), medium (M), and high (H) load scenarios. The text in parantheses in the first row indicates the type of query which was used in the experiment. The numbers in parantheses in the second row indicate the number of simulated stations in the experiment. One would expect the final result to arrive immediately after the last triple was streamed to the engine, which equals the duration of one test, i.e., 30 seconds in our case. This is the case when we put CQELS under low load as shown in the columns denoted by an *S*. However, under medium and high load, denoted by *M* and *H* columns, we see that delivery delay of the results grows.

	E1 (SELECT)			**E2** (AVG)			**E3** (SELECT + JOIN)		
	S (50)	**M** (1000)	**H** (10000)	**S** (50)	**M** (1000)	**H** (10000)	**S** (50)	**M** (1000)	**H** (10000)
AVG	30s	32s	98s	31s	432s	12966s	31s	62s	269s
MIN	30s	32s	97s	31s	426s	12305s	31s	58s	263s
MAX	30s	33s	100s	31s	440s	13523s	31s	65s	277s

Table 3.5: Arrival time (average, minimum, maximum) of final results in seconds for each conducted experiment (E1, E2, E3) with CQELS

The reason for that is that CQELS uses the *OnContentChange* report strategy, where queries are evaluated after each streamed triple. Obviously, more complex queries increase the time needed for computation of query results, resulting in a monotonous increase of delay as is also shown in Figure 3.16c. Hence, the arrival of the last result helps to quantify and infer the influence of different query types on the capability of CQELS to provide timely results. These numbers should not be compared with C-SPARQL where such delays did not appear due to its report strategy In C-SPARQL queries are only evaluated when a window closes which results in much lower computational effort and hence avoids high delays in delivering results.

3.7 Summary

In this chapter, we introduced YABench, a comprehensive RSP benchmarking framework that provides detailed insights into the performance and correctness characteristics of RSP engines based on granular metrics, including metrics that capture the capability to produce correct results under load. YABench is a general framework which allows the flexible definition of streaming scenarios. We defined an environmental stream data scenario simulating measurements of environmental sensors. The combination of the dynamic generation of streaming data with the extended set of evaluation instruments offered by YABench, enables to benchmark state of the art RSP engines with respect to environmental stream data. The results and detailed discussion of running four experiments lead us to the conclusion that, compared to CQELS, C-SPARQL is the preferable approach in our scenario.

The framework is able to carry out the complete benchmarking process from defining tests, generating suitable test data, executing tests, and finally analyzing the results. We have shown that the framework replicates basic results of an existing benchmark (i.e., CSRBench) and conducted and discussed four more comprehensive experiments. The resulting visualizations provide insightful information on the characteristics of the tested engines and highlight key differences in their performance. In the process of our benchmarks, we also identified and extensively discussed previously unknown issues.

To sum up our findings, YABench reveals that C-SPARQL operates more memory efficiently than CQELS in all experiments. Concerning delay, both engines perform similar. However, the use of more complex queries and an increase in input load lets C-SPARQL outperform CQELS. The main reason is the report strategy of CQELS where queries are re-evaluated whenever new data arrives, whereas C-SPARQL only evaluates queries when an active window closes which, obviously, happens less frequently. Concerning precision and recall CQELS yields better results for simple `SELECT` queries. However, under certain circumstances (joining triples of one window) we detect an issue in CQELS which results in decreasing precision and recall measurements. On the other hand, C-SPARQL suffers from window delays, which increase when load on the engine is raised. These delays are responsible for declining precision and recall. A *gracious* mode to run the oracle, allows us to reveal the extent of these effects visually.

CHAPTER 4

Environmental Streaming Mashups

After describing and discussing a data model for environmental stream data in Chapter 2 and presenting the implementation of a framework for evaluating RSP engines in Chapter 3, we present the implementation of an architecture based on foregoing results.

The architecture connects the concepts of the environmental data model with semantic stream data and widget-based mashups and is presented in detail in the following sections. The platform on which the architecture is prototypically implemented is the "Linked Widgets platform"[1] [TDW+13, TWD+14b, TWD+14a, TWD+14c, TWD+15]. This platform facilitates data exploration and we extend it to enable processing of stream data. We introduce stream processing mechanisms embodied in Linked Streaming Widgets. The framework is based upon semantic annotations that describe the data using domain vocabularies that can be used to integrate heterogeneous environmental data. "Linked Streaming Widgets" represent an extension of "Linked Widgets" which are lightweight web applications equipped with a semantic model that can be used to compose mashups.

We start with the presentation of related work in environmental data integration (Section 4.1). To become familiar with the underlying concepts, ideas, and features of the platform we present its central aspects (Section 4.2). Then, we describe two use cases to demonstrate the feasibility of our extensions to the Linked Widgets Platform (Sections 4.3 and 4.4). Finally, we conduct a performance evaluation of the approach (Section 4.6).

4.1 Related Work

Data integration — the problem of combining data residing in different sources — has been a field of interest for computer scientists for years [Len02]. After database technology

[1] `http://www.linkedwidgets.org` (accessed 16 June 2016)

has been introduced in companies in the late 1960s, the number of data repositories increased rapidly which revealed the need for data integration [ZD04].

First integration approaches were based on multi-database systems in the 1980s [HB91, LR86]. Consecutive approaches to tackle the issue of data integration used mediators [CHS+95], agent systems[BJBB+97], and, more recently, ontologies [MIKS00], peer-to-peer [AKK+03], and web services [ABM02].

In the context of web services, which are used for "deploying automated interactions between distributed and heterogeneous applications" [BDS08], a new concept called *mashups* emerged. Mashups deal with the issue of data integration on the web by creating web applications through combining existing web resources (e.g., data or APIs). The concept has been used to solve data integration challenges. A novel aspect of mashup development is that users do not necessarily need programming expertise in order to create these applications; the aim is to reduce the technical knowledge required by the target group. This is achieved, for instance, by guiding users through the process of mashup creation [LHPB09].

We perform an analysis of existing environmental data mashup systems with respect to a set of requirements. The requirements we identify are drawn from the scenario of supporting the composition of mashups based on environmental stream data. They are considered to be relevant not only for the environmental application domain, but also for a broad range of other domains. We discuss each mashup system in detail and provide a discussion of shortcomings of the state of the art.

R1 – Semantic Stream Data Processing. The mashup system should be capable of processing semantic stream data sources of interest in order to provide results in real time and to be able to make use of the semantics inherent to the data.

R2 – Data Integration. It should be possible to integrate stream data sources with static background knowledge, such as sensor metadata within a mashup. This allows to contextualize the data generated by environmental sensors and to gain novel insights based on the integrated data.

R3 – Composition Support. Users should be supported in composing mashups in that they compose syntactically and semantically valid solutions. Often it is difficult for users to know which mashup modules can be combined. This lack of support maximizes composition errors which become visible during execution of the final mashups.

R4 – Discovery Support. Users should be supported in discovery of data sources and mashup modules. This makes it easier for users to identify interesting data and to retrieve modules which again can be combined with respective data sources.

R5 – Visual Programming. The mashup system should follow the visual programming approach. This allows users without any programming background to create their own mashups.

R6 – Alerting Support. In the domain of environmental stream data, monitoring use cases are of special interest. Often it is necessary to get notified by the system, if observations are retrieved which are above or below a certain threshold. Hence, the system should support means for alerting or notification.

R7 – Extensibility. The system should support adding new environmental data sources and provision of new modules which can be used by other mashup developers via a Graphical User Interface (GUI). This increases the system's flexibility and ensures that the system will stay up to date.

4.1.0.1 Videk

Videk [KFFG11, KFM+13] (cf. Figure 4.1) is a prototype mashup for environmental intelligence. The system allows to visualize environmental sensor data on a map. Users can follow current observations or view historical measurements aggregated over different time periods. The data is enriched with data from external sources (e.g., Geonames, Wikipedia, Google Maps) to help users interpret the measurements.

Videk consists of four main components, (i) the sensor data, (ii) the mashup server, (iii) the external sources, and (iv) the user interface. The mashup server forms the core, because it interfaces with all other components. It combines raw and processed data with external sources and provides a GUI for end users. Further, the mashup server contains a storing and processing engine called SenseStream which transforms the sensor

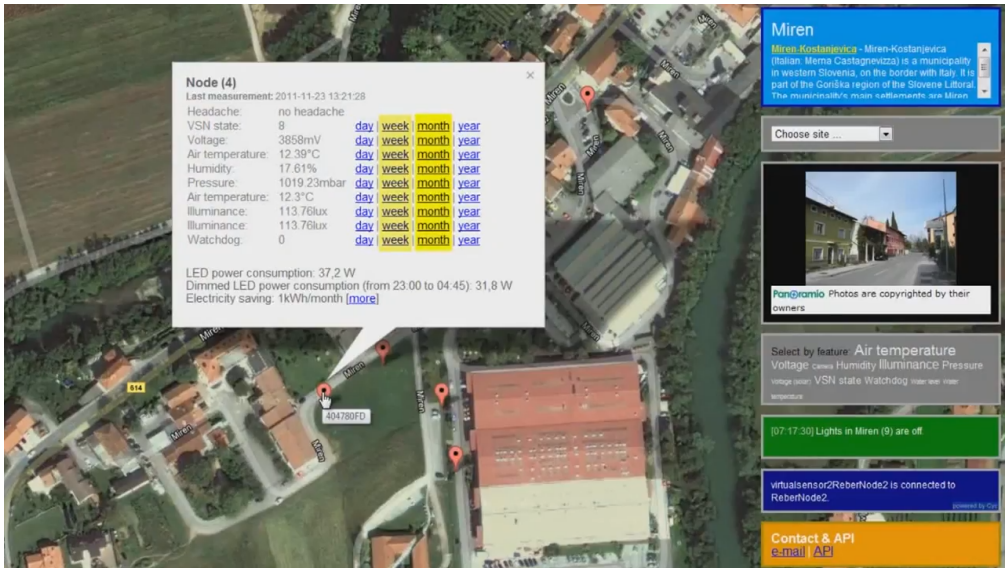


Figure 4.1: Screenshot of Videk[2]

[2]Screenshot taken from `https://www.youtube.com/watch?v=p5HJC%_af9UY` (accessed 16 June 2016)

data into useful information and knowledge. Generally, the architecture allows to add new external and sensor data sources. SenseStream also supports addition of plug-ins to add features to the processing pipeline. The GUI uses an API layer to retrieve sensor metadata as well as recent measurements. The interface is based on the Google Maps layer and enables browsing and exploring the available sensor nodes.

We analyze Videk regarding our defined requirements as follows. Videk is able to process stream data (R1) by integrating QMiner which is a stream mining and event detection engine. However, it is not clear how the authors adapt QMiner in order to be able to process semantic stream data, or if it processes semantic data at all.

The optional *knowledge base* component of Videk can store static information for event processing. However, the component is primarily used to generate rules and not to build mashups. We therefore evaluate R2 as partially fulfilled.

Users are not supported (R3) in creating mashups with Videk. In fact, Videk itself is a mashup which has been developed by programmers. The GUI contains widgets, such as a location widget and a photo widget, but they cannot be used to integrate available data.

The selection widget of Videk allows users to toggle the visibility of sensor nodes according to their measuring capabilities. Hence — at least to some extent — users are supported in discovering data sources (R4).

Videk does not follow the visual programming approach (R5). It does not help users that lack programming experience in creating new mashups.

Despite Videk being aimed to process sensor measurements, it does not provide any means to be notified, if measured values meet certain criteria, such as exceeding a threshold (R6).

Means to extend Videk with new data sources and widgets were not identified. Hence, extensibility (R7) is not supported. However, an API is offered which allows the system to be plugged into other external mashups.

4.1.0.2 Traffic LarKC

Traffic LarKC [DCDV13, DVCD+11] (cf. Figure 4.2) is a mashup application that integrates traffic data to forecast street conditions and to find optimal routes in a city. To this end, the application combines information from remote sources, that is, maps, traffic sensors, weather data, and calendar data. The mashup uses RDF as an interchange format to support the integration with semantic processing and reasoning.

The main challenges of Traffic LarKC in providing traffic predictions is the volume, quality, and heterogeneity of the data. According to the authors, the traffic database contains more than one billion triples, however, due to the open world assumption [Kee13], many effects are not observed, such as parking cars or minor accidents.

Traffic LarKC does not process semantic stream data (R1). Even though the authors state that sensor data of 300 traffic sensors is processed, the system only operates over

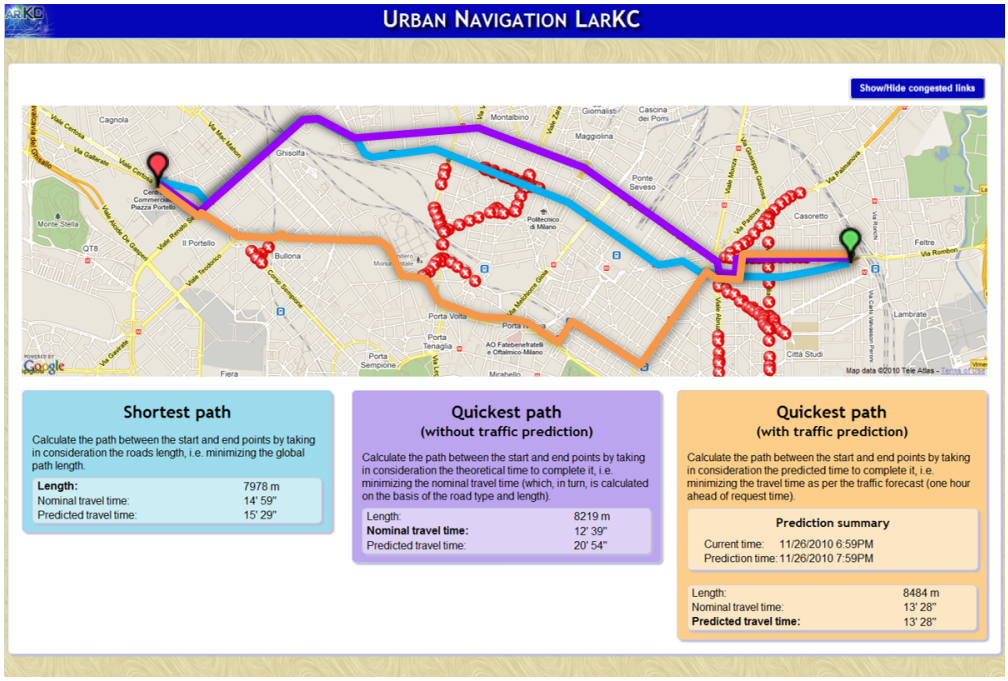


Figure 4.2: Screenshot of Traffic LarKC[3]

archived data. This indicates that real time results are not provided which, however, would be expected if stream processing is performed.

The authors show that Traffic LarKC integrates data from different sources (R2). Even if the mashup does not explicitly process stream data, data as different as traffic sensor data, weather data, or event data are integrated.

Traffic LarKC does not allow end users to compose their own mashups (R3). This seems not to be the primary aim of this application because it has been designed for the purpose of providing traffic predictions and route recommendations.

The aim of Traffic LarKC is not to make data sources combinable in a flexible manner for users. Hence, users are not able to discover new data sources or mashup modules (R4). Similarly, visual programming (R5) and a notification service (R6) are also not provided to users.

Traffic LarKC is extensible in that it is built upon the The Large Knowledge Collider (LarKC) platform which offers a pluggable architecture that enables to integrate new plug-ins into existing workflows and components, such as Traffic LarKC (R7). Because this does not mean that the mashup application per se is extensible, we evaluate this requirement as partially fulfilled.

[3]Screenshot taken from [DCDV13] (Original file kindly provided by Irene Celino (`http://orcid.org/0000-0001-9962-7193`).

4.1.0.3 SensorMasher

SensorMasher [LPH09, LP09, LPPH+10] (cf. Figure 4.3) is a mashup infrastructure to publish stream sensor data. It provides a user interface to explore sensor data and build web mashups. Its main goals is to enable non-technical users to access and manipulate sensor data in an intuitive fashion. The platform is based on RDF data. However, the semantic descriptions and annotations of raw sensor readings and sensors need to be done by the user. These semantic descriptions can then be exploited in mashups and in linked open data scenarios in order to enable the discovery and integration of sensors and sensor data.

The architecture of SensorMasher consists of a DSMS, query processor, Sensor&Mashup manager, user manager, explorer, composer, and web interface. New sensor data is pushed to the DSMS. The query processor evaluates continuous queries over the data in the DSMS. The Sensor&Mashup Manager controls the data flow between the DSMS and the fusion operators, which are used for processing operations such as data filtering and data alignment. It is also used to edit and query metadata, deploy mashups, and provide access control via the user manager. The explorer component enables users to explore sensor data on a map via a GUI. The composer is used for visually composing sensor mashups by connecting sensor sources with fusion operators, and finally, the web interface includes a web services interface and a SPARQL endpoint.

SensorMasher processes semantic stream data and provides answers in real time via queries running over a DSMS (R1). Similarly, the infrastructure enables integration of sensor data following the linked data principles (R2). The data is first published and can then be integrated with other (linked) data sources via mashups.

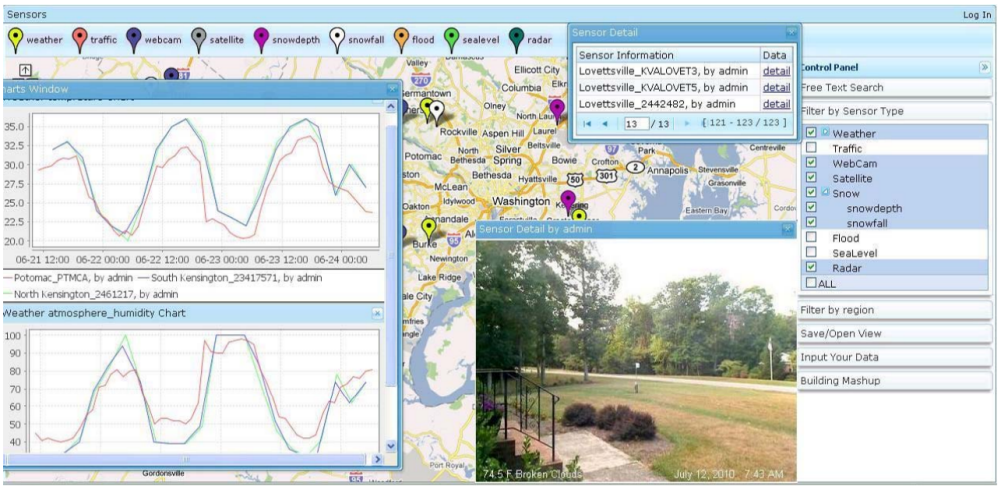


Figure 4.3: Screenshot of SensorMasher[4]

[4]Screenshot taken from [LPPH+10].

The provided visual composer component allows users that lack programming knowledge to create mashups (R3). The composer works similar to a workflow editor where data sources and operators can be connected to create new sensor mashups. However, it does not provide means to semantically support this process in order to prevent erroneous compositions. In addition, the interface has been criticized in that "the process is very difficult and not user friendly" [LH14].

Data source discovery is also enabled via the composer component (R4). It provides means to navigate and explore sensors which can then be used to compose mashups. The visual composer is also responsible for implementing the visual programming paradigm (R5).

Instruments to create alerts when users want to observe sensor measurements in mashups are not available (R6), despite monitoring being mentioned as a motivating scenario by the authors.

According to the authors, new sensor data sources can be added, however, it is not explained how this works (R7). Sensor data which is published by SensorMasher can be accessed by external applications through a SPARQL endpoint and RESTful web services.

4.1.0.4 SemSorGrid4Env

SemSorGrid4Env [GGK+11, GSK+11] (cf. Figure 4.4) is the implementation of a service architecture to support environmental decision making. The main use case is flood emergency response. The architecture enables to put sensor readings into context with other sensor readings, sensor data histories, and stored data. The overall goal of SemSorGrid4Env is to enable ad hoc generation of mashups over data from computations combining real time and historical data.

The architecture is divided into three tiers, i.e., a data tier, a middleware tier, and an application tier. The data tier enables the publication and querying of data through a *connectivity bridge*. The middleware tier supports the discovery of data, reconciliation of data models, and querying over these models. The *semantic integrator* and *semantic registry* are the two main components in this tier. Finally, the application tier provides domain specific services and REST services to applications and mashups.

SemSorGrid4Env processes stream data in order to provide real time results to users (R1). The authors use $\text{SPARQL}_{\text{Stream}}$ [CCG10] to support ontology-based data access for stream data.

Given that correlation of data with different modalities and heterogeneous data integration are two main features of SemSorGrid4Env, it satisfies our data integration requirement (R2). This is also shown in the example application of the flood emergency response mashup.

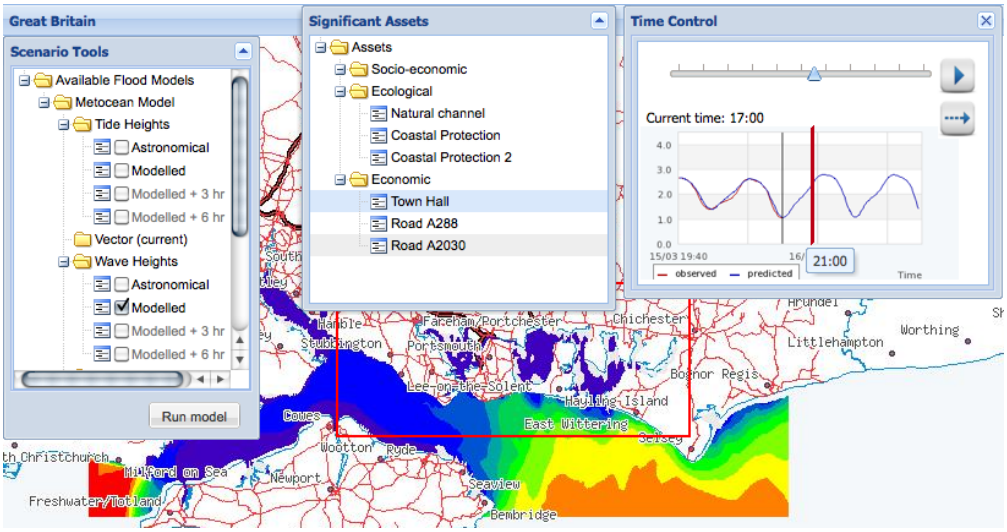


Figure 4.4: Screenshot of SemSorGrid4Env[5]

Users are not supported in composing new mashups (R3). The system provides REST interfaces which can be used by web application developers, but not by potential end users who lack programming expertise.

Discovery of new data sources is enabled by the semantic registry component and the publication of semantically annotated property documents (R4). However, this is only enabled for other services and not for human users via a user interface. Similarly, the presented system does not implement a visual programming approach (R5). The presented use case of flood emergency response is a web application which has been built on top of the architecture. The authors state that the architecture enables the creation of mashups, however, they do not provide a visual interface to do this.

The architecture enables the definition of events based on sensor measurements. When conditions that characterize such an event are identified, alerts are raised (R6). In the example scenario such an event is triggered when waves go over the top of a sea defence.

SemSorGrid4Env (re)uses technological standards, such as REST services, Web Services Description Language (WSDL), and well-known ontologies (R7). This ensures extensibility on a technical level, but does not allow end users or mashup developers to easily add new sources or modules, hence, only partially fulfills this requirement.

4.1.1 Summary

Table 4.1 shows a comparison of the presented environmental mashup systems with respect to the defined requirements. A checkmark (✓) denotes *full support*, a hyphen (-)

[5]Screenshot taken from [GSK+11] (Original file kindly provided by Alasdair J G Gray (`http://orcid.org/0000-0002-5711-4872`).

denotes *no support*, and a tilde (~) denotes *partial support* of the respective requirement.

Videk and Traffic LarKC do perform stream processing of data. Each supports only one of the requirements which we identified for environmental data mashups to full extent. Videk supports discovery of data sources and Traffic LarKC supports data integration. SensorMasher and SemSorGrid4Env both can deal with stream data sources and enable data integration. They also support some of the remaining requirements. We show that existing systems still severely lack features which we consider as substantial for environmental data mashup systems. The work presented in this thesis fills this gap and contributes by providing an approach that satisfies the specified requirements.

Mashup	**R1**	**R2**	**R3**	**R4**	**R5**	**R6**	**R7**
Videk	~	~	-	✓	-	-	-
Traffic LarKC	-	✓	-	-	-	-	~
SensorMasher	✓	✓	-	✓	✓	-	~
SemsorGrid4Env	✓	✓	-	~	-	✓	~

Table 4.1: Environmental data mashup systems comparison

4.2 Linked Streaming Widgets

Data plays an increasingly important role in our everyday life. Collecting data from different sources and extracting information, however, is still difficult due to the abundance of available data. To enable users to benefit from published data, we need to overcome different data integration challenges:

(i) Data heterogeneity makes it difficult to integrate different types of data available in different formats distributed among infrastructures.

(ii) Cumbersome manual data integration processes that users perform to collect, clean, enrich, integrate, and visualize data are neither reproducible nor reusable.

(iii) Lack of support for exploration, as users often rely on domain-specific applications that do not allow to integrate arbitrary data sources.

(iv) Lack of means for the identification of relevant data sources and meaningful ways to integrate them.

End users currently are not able to fully realize the potential of available data. Instead they have to rely on applications built by programmers. The concept of End User Programming aims to emancipate users from this dependency on programmers. It allows them to build up an application within a short amount of time [Mar95]. In the research field of End User Programming, widget-based mashups implement the visual programming

paradigm which allows end users to compose ad hoc applications by combining available widgets and thereby integrate data from disparate sources. Such applications use "content from more than one source to create a single new service displayed in a single graphical interface" [Eng15] to increase the value of previously existing data.

The Linked Widgets platform follows a widget-based mashup approach. It aims to (i) provide practical advantages in data processing without imposing restrictions on data sources, (ii) allow users to combine multiple data sources leveraging their joint value, and (iii) allow end users to analyze, integrate and visualize data. This enables urban stakeholders to facilitate environmental stream data to make well-informed decisions.

The platform uses semantic web technologies and its design follows three guiding principles, namely "openness", "connectedness", and "reusability". Openness is the key to achieve our first objective, i.e., the capability to deal with heterogeneous data sources. Developers can therefore implement and directly add new widgets to the platform. Connectedness means users can combine data from different sources by connecting different widgets possibly curated by different developers. Finally, the ability to use the same widget in a flexible manner to compose applications that serve different purposes, facilitates reusability.

We use a graph-based model (cf. Section 4.2.1.1) to semantically describe the input and output of a widget so that the platform can make use of the annotated models to provide semantic search (cf. Section 4.2.1.4), terminal matching (cf. Section 4.2.1.5), and auto composition (cf. Section 4.2.1.6).

4.2.1 Linked Widgets Platform

In this section we provide a general overview of the Linked Widgets Platform including the underlying data model, a characterization of available widget types, and provided features and extensions by the platform.

4.2.1.1 Linked Widgets Data Model

Linked Widgets [TDW+13] extend standard widgets with a semantic model following linked data principles. The semantic model describes data input/output and metadata such as provenance and license. In particular, the model consists of four main components: (i) *input terminals*, (ii) *output terminals*, (iii) *options*, and (iv) *a processing function*. Input/output terminals are used to connect widgets in a mashup and represent the data flow. Options are HTML inputs inside a widget. They provide a mechanism for users to control a widget's behavior. Finally, the processing function defines how widgets receive input and return their output.

We distinguish three types of widgets: *data widget*, *process widget*, and *visualization widget* (cf. Section 4.2.1.2). A mashup is an interconnected combination of widgets. It should contain at least one data widget providing the data and one visualization widget to display the final results.

Our widgets input and output terminals are enriched with semantic models. These semantic models are essential for the subsequent search and composition processes. Furthermore, they are crucial for the effective sharing of widgets. Discovering appropriate widgets for one's environmental information need is a non-trivial task which our model facilitates. For instance users may be interested in pollution data. Hence, the semantic model supports users in finding appropriate data sources and in finding widgets which can be used in the composition process. Existing mashup platforms usually employ a text-based approach for widget search, which is not particularly helpful for advanced widget exploration and widget composition tasks.

Figure 4.5 presents a part of our ontology for the modeling of Linked Widgets. The use of semantic web technologies to describe mashups and their components is not by itself a novel approach (see [NCSP10, PRM11]). However, rather than capturing the functional semantics and focusing on input and output parameters like SAWSDL [KVBF07], OWL-S[6], and WSMO[7], we use a graph-based model [TKSA12, TKSA13, VSVD+11] to formally annotate the input and output components as well as their relations. The *SWRL* vocabulary is reused to define the semantic relation between two nodes in the input and output graphs. This is shown in the bottom of the figure by the representation of an *extra relation*.

Figure 4.5 also shows the detailed model of an exemplary *Geo Merger* widget. The widget takes two arrays of arbitrary objects containing the *wgs84:location* property as input. Its domain is the *Point* class with two literal properties, i.e., *lat* and *long*. The widget output is a two-dimensional array in which each row represents two objects from two input arrays, respectively. Those objects include locations satisfying the distance filter of the *Geo Merger* widget.

To specify that input/output is an array of objects, we use the literal property *hasArray-Dimension* (0: single element; $n > 0$: n-dimensional array). Because the input of *Geo Merger* is an "arbitrary" object, we apply the *owl:Thing* class to represent it in the data model.

The *point*, *location*, *lat* and *long* terms are available from different vocabularies. However, since a well-established ontology facilitates data exchange between widgets, we chose *wgs84*. The *widget annotator* module (cf. Section 4.2.1.3) interactively recommends frequently used terms of the most popular vocabularies to developers. This eases the annotation process and fosters consistency by diminishing the use of varying terms to describe the same concepts.

In addition, we can model more advanced widgets by describing additional relations connecting the input and output of a widget. Namely, the input and output can consist of object attributes which can be connected with an added relation. As depicted in the bottom section of Figure 4.5, we create a *nearby* relation between the two input points.

[6] `http://www.w3.org/Submission/OWL-S/` (accessed 16 June 2016)
[7] `http://www.w3.org/Submission/WSMO/` (accessed 16 June 2016)

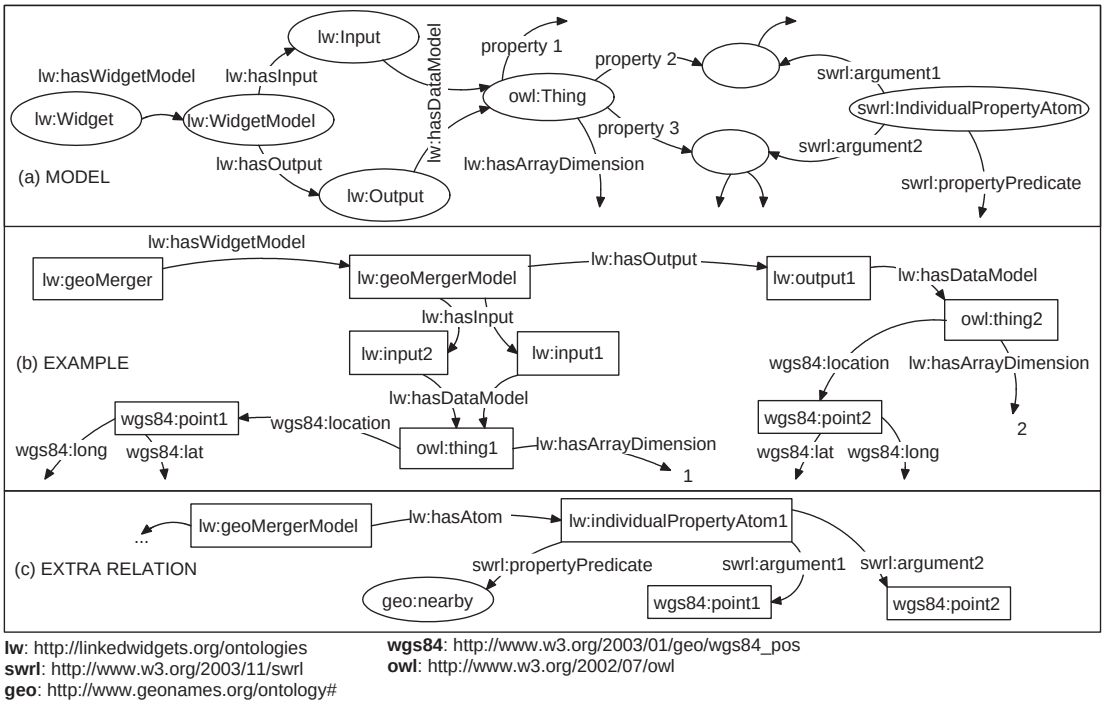


Figure 4.5: General Linked Widget model and Geo Merger model

Due to this graph-based description, the platform can answer questions such as “find all widgets that contain the *nearby* relation between two locations".

4.2.1.2 Characterization of Linked Widgets

Linked Widgets can essentially be characterized along two dimensions, that is, location of execution (client vs. server widgets) and type of operation (data vs. processing vs. visualization widgets) Both characterizations will be described briefly in the following.

Location of execution A novel aspect of the Linked Widgets concept is that they can be executed in distributed environments. Mashups can be composed of both client and server widgets. “Client widgets” are executed in the local context of a Web browser environment using client memory and processor resources. The data is collected and processed on-the-fly in the browser.

Each *client widget* consists of:

- a semantic model,
- an execution function,
- input and/or output terminals,

- a core widget interface which is automatically generated for users to run the widget,
- its own user interface programmed by developers.

The execution function transforms the received input into an output according to the parameters specified in the interface.

To implement a *client widget*, developers create a user interface and follow three steps:

1. inject a JavaScript file[8] to enable cooperation with other widgets,
2. define an input and/or output configuration, and
3. implement a JavaScript `run(data)` function that is invoked when the widget is executed in a mashup.

Client widgets are easy to develop, however, their capabilities are restricted by the web browser execution environment. Furthermore, they cannot deal with heavy data processing tasks. Finally, as soon as a user closes the browser, the mashup output data can no longer be accessed. To overcome these limitations, we designed *server widgets* in order to shift the execution function from the browser environment to standalone application environments. Server widgets can be executed as native applications on different kinds of platforms such as servers, mobile devices, or embedded systems. These platforms can possibly serve as sources for streaming data.

These server widgets consist of two main parts:

- a *user interface*, which is the same as for *client widgets*, but without the *run* function, and
- a *remote executor*.

The *client interface* is for users to set up parameters and control the *remote executor*.

When users drag-and-drop a *server widget* into the mashup editor panel, a *client user interface* of the *server widget* is instantiated. At the same time, we set up a connection channel between this *client interface* and the *remote executor* of the *server widget*. For each *server widget*, we have only a single *remote executor*, but potentially many *client user interfaces* that are instantiated for each instance of the widget. When an instance of a *server widget* is executed, the *client interface* sends its parameters to the *remote executor*, which, in turn, creates a *widget job* to process the data received from the predecessor widgets.

Server widgets make use of the concept of *distributed mashups* – a type of ad hoc applications whose processing tasks are executed in a distributed manner possibly on

[8] `http://linkedwidgets.org/widgets/WidgetHub.js` (accessed 16 June 2016)

multiple devices. Such mashups are particularly useful for streaming or real time data processing applications. Users can close the browser at any time while the backend performs data collection and processing tasks.

There are two sub-types of *server widgets*, i.e., *server data widgets* and *server processing widgets*. Servers are the most suitable targets to deploy (server) *processing widgets* because they are continuously online. This also allows to offload computationally intense data processing tasks from mobile devices. Mobile devices, however, are ideal environments for (server) *data widgets*. They can, for instance, collect and provide data from mobile devices for a mashup. For example, smartphones can act as sensors that periodically provide Global Positioning System (GPS) data.

To create *server widgets*, developers first define the *client interface* in a similar manner to the *client widgets* implementation. Furthermore, they need to build the *remote executor* component. To this end, they implement an abstract method to define the widget job, i.e., a processing function to transform input data into output data.

Type of operation There are three types of widgets: *data widget*, *process widget*, and *visualization widget*. A *data widget* collects data from a data source and provides the collected data semantically enriched to other widgets. It has no input terminals. A *process widget* takes input data from other widgets, applies operations on the data (enrichment, transformation, and aggregation), and provides the result to consecutive widgets. It consists of both input and output terminals. A *visualization widget* has at least one input terminal and presents the data from another widget in a particular manner (e.g., textually or visually). It has no output terminals.

4.2.1.3 Annotator

The *widget annotator* allows developers to create and annotate widgets correctly and efficiently according to our defined semantic model (cf. Section 4.2.1.1). Developers can use the drag-and-drop editor. In order to visually define their widget models, they need to configure three components called the *Widget Model*, *Object*, and *Relation*. The final output of the annotator is an RDF-based semantic model describing the input and output terminals of a widget. This output is then stored together with the source code of a widget and enables the platform to provide features, such as widget search, terminal matching, and auto-composition.

Figure 4.6 is an example that illustrates the definition of a semantic model for a *Citybike Station Filter* widget. In the *Widget Model*, we define input and output terminals of the widget. Their data models — arrays of *CityBike* data at the output and *Place* objects at the input — are defined in the *Object* components. We define property URIs for the *CityBike* type, such as *freeBikes* and *freeBoxes*. Among these properties is also a *wgs84:location* property. Its domain is defined via an additional *Object* component. The input is a *Place* which also contains a *wgs84:location*. Finally, we make use of a *Relation* component to specify *geo:nearby* as a relation between the input *Place* and the output

CityBike objects. This means that the widget outputs data about citybike stations which satisfy the *nearby* requirement with respect to the *Place* at the input.

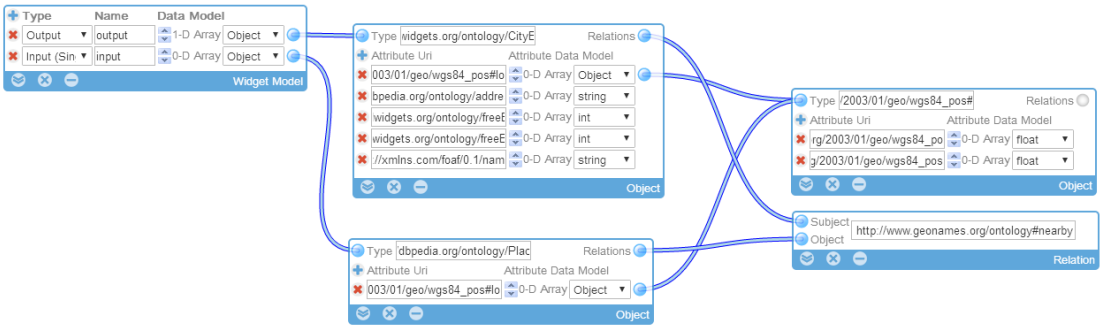


Figure 4.6: Visual model defined for the Citybike Station Filter widget

The system then automatically generates the OWL description file for the model as well as the corresponding Hypertext Markup Language (HTML) widget file. It includes the injected JavaScript code snippet required for the widget communication protocol and a sample of the input/output of the widget according to the defined model. Based on the HTML file, developers can implement the processing function of the *client widget* or the *remote executor* of the *server widget*, which receives input from preceding and returns output to succeeding widgets. Widget annotations are published into the Linked Open Data (LOD) repository of widgets which can be accessed via a SPARQL endpoint[9].

4.2.1.4 Semantic Widget Search

To cope with a growing number of available widgets on the platform we provide a semantic search feature in addition to conventional search methods which are based on keywords, categories, and tags. Our semantic search works as follows: After defining desired class, property, and relation constraints for input/output, the platform formulates a SPARQL query which is then executed over the available widget repository in order to return widgets satisfying defined requirements. To stick to the previous example, for instance, we can find widgets which return *CityBike* data containing the *freeBikes* attribute and have a geographic *Place* as an input. The generated and executed SPARQL query returning the desired widgets is shown in Listing 4.1. In this case the *Citybike Station Filter* widget is returned.

4.2.1.5 Terminal matching

A common task when users create mashups is to find widgets which can be connected to another widget. We call this procedure *terminal matching*. It is crucial to connect only outputs with inputs if the data which is transmitted along this connection can be

[9] `http://ogd.ifs.tuwien.ac.at/sparql` (accessed 16 June 2016)

```
PREFIX lw: <http://linkedwidgets.org/ontology/>
PREFIX wgs84: <http://www.w3.org/2003/01/geo/wgs84_pos#>
PREFIX dbpedia: <http://dbpedia.org/ontology/>

SELECT DISTINCT ?widget WHERE
{
        ?widget lw:hasWidgetModel ?widgetModel.
        ?widgetModel a lw:WidgetModel.
        ?widgetModel lw:hasInput [lw:hasDataModel ?iDataModel].
        ?iDataModel a dbpedia:Place.
        ?iDataModel wgs84:location . [
                a wgs84:Point;
                wgs84:lat [];
                wgs84:long []; ].
        ?widgetModel lw:hasOutput [lw:hasDataModel ?oDataModel].
        ?oDataModel a lw:CityBike .     [
                lw:freeBike []; ]
}
```

Listing 4.1: A SPARQL query for semantic widget search

processed by the connected widget. To enforce valid connections (i.e., ensure that the output terminal can provide all data required at the input terminal), mashup developers can use the terminal matching module. This module validates connections using the semantic model. It helps creators to speed up the mashup creation process, while preserving syntactic and semantic correctness of a mashup. Listing 4.2 shows a SPARQL query detecting output terminals which can be connected to the input terminal of the *Citybike Station Filter* widget.

The following conditions need to be satisfied for matching terminals:

- matching class and array dimension,
- matching attributes, i.e., the set of attributes required by the input terminal must be a subset of the attributes provided by the output terminal.

4.2.1.6 Automatic mashup composition

The automatic mashup composition algorithm incorporates the semantic model and the terminal matching algorithm to allow for full-automatic composition of mashups. Users may lack required knowledge about how to define and control the widget flow during composing a mashup. When confronted with new widgets, users may consider the process to create a mashup including a data widget, processing widgets, and finally a visualization widgets tedious and time consuming. We therefore support them by assembling complete mashups automatically from a set of available widgets. Essentially, this approach enables automatic composition of a complete mashup from a widget that consumes/provides data for a specific output/input terminal. “Complete” in this context means that all terminals must be wired, i.e., have a valid connection.

```
PREFIX ifs: <http://ifs.tuwien.ac.at/>
SELECT DISTINCT ?oTerminalName ?oWidget WHERE {
  <http://ifs.tuwien.ac.at/WidgetCityBikeStationFilter>  ifs:hasWidgetModel  ?iWModel.
  ?iWModel ifs:hasInput  [ifs:hasName "input1"^^xsd:string;
                          ifs:hasDataModel ?iDataModel].
  ?iDataModel a ?type.
  ?iDataModel ifs:hasArrayDimension ?listLevel.

  ?oWidget ifs:hasWidgetModel  ?oWModel.
  ?oWModel  ifs:hasOutput [ifs:hasName ?oTerminalName;
                           ifs:hasDataModel ?oDataModel]
  ?oDataModel a [rdfs:subClassOf ?type].
  ?oDataModel ifs:hasArrayDimension ?listLevel.

  FILTER NOT EXISTS{
    ?iDataModel ?property ?iValue.
    FILTER NOT EXISTS {?oDataModel ?property ?oValue.}
  }
}
```

Listing 4.2: A SPARQL query for terminal matching

The fundamental steps of this approach are as follows: We first construct a weighted graph based on terminal matching for a given set of widgets. The vertex set consists of all input and output terminals of all widgets. Edges of the graph represent valid connections between terminals. Next, we have to identify complete paths in this graph starting from all output and input terminals of the widget such that all involved input/output terminals are wired with other output/input terminals. Each identified complete path represents a valid mashup solution. More details on the algorithm are presented in [Tri16].

As an example, we provide a collection of eight widgets to present the citybike use case (cf. Section 4.3). The automatic mashup composition algorithm is initiated when users select the output terminal of a data source widget such as the *Map Pointer*. The algorithm then builds one graph containing all widgets of the collection and detects complete paths in this graph. An excerpt of the graph is depicted in Figure 4.7. For the citybike widget collection the algorithm then detects 19 semantically and syntactically valid complete paths, i.e., mashups, each consisting of two to five widgets.

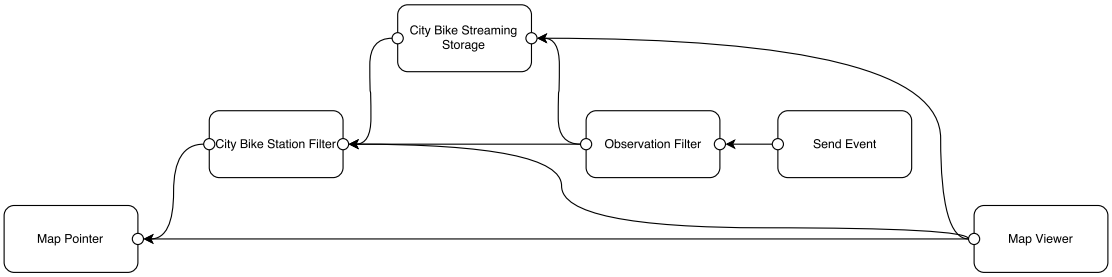


Figure 4.7: Graph used by the automatic composition algorithm to find complete mashups

4.2.2 Streaming Widgets Architecture

We present RDF Stream Processing (RSP) extensions to the Linked Widgets platform architecture in order to cope with stream data. RSP is an emergent field of research [VCHF09] with the aim to facilitate data integration on real time semantic data streams. This paradigm is similar to the fundamental concepts of the Linked Widgets Platform in that both focus on the integration of semantic data. The main requirements to realize this paradigm are (i) to enable the use of stream data in mashups in order to gain real time insights based on integrated data and (ii) to provide a means to integrate different sensor sources among each other, but also to integrate them with static background knowledge sources. We present extensions to the platform that foster the processing of RDF streams by means of widgets.

Our analysis of RSP engines (C-SPARQL and CQELS) in Section 3, leads us to the conclusion that C-SPARQL is the preferable system to be used for linked streaming widgets. The main reason for this decision is the fact that C-SPARQL reports new results based on a *Window Close* policy. This means that results flow into the system at regular intervals facilitating further processing such as aggregation functions. Moreover this also fits with the use cases we realize based on environmental data sources where new measurements are taken also at regular intervals.

Additionally, the fact that C-SPARQL allows for building query networks, where the results of queries can be reused in other queries facilitating aggregation and reducing processing load represents further advantages. Finally, C-SPARQL also shows better performance over CQELS in terms of memory consumption and Central Processing Unit (CPU) load.

Figure 4.8 shows the overall architecture and processing model of the streaming extensions to the Linked Widgets platform. As shown at the very left of the figure, arbitrary data sources such as XML, JSON, and CSV files can serve as input. These data sources are then RDFized[10], i.e., converted, into RDF triples via RDF Mapping language (RML) mappings [DSC+14, DSS+14]. RML mappings only need to be created once per data source. We slightly extend the standard features of RML and allow to use variables in the mapping definitions which are replaced by desired values at processing time.

The RML mappings use special domain ontologies as well as the SSN ontology [CBB+12] and RDF Data Cube vocabulary [CR14] to create triples conforming to the ontology developed in Section 2. We present use cases in the following sections, which apply the RML mappings in regular intervals on the respective data sources.

The semantically enriched data sources are then sent regularly to an instance of the C-SPARQL server[11]. We prepare streams at the C-SPARQL server for each data source. Hence, the triples of each RDFized data source are sent to the accompanying stream at

[10] the term *RDFizing* is commonly used in the semantic web literature to refer to the transformation of arbitrary data to semantically enriched RDF data. Similar terms are *triplifying* or *semantically lifting*.

[11] C-SPARQL provides a REST API which is used here [BV13].

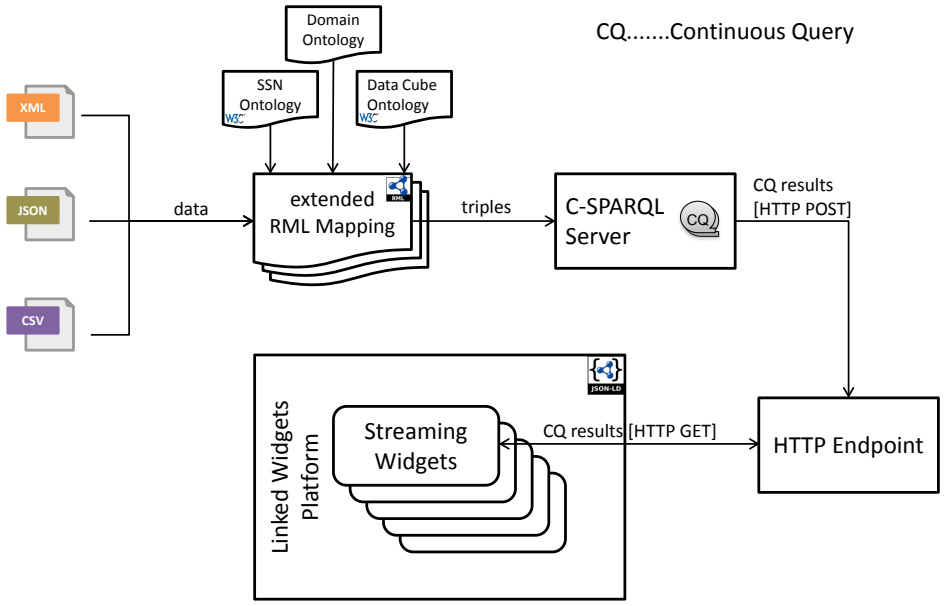


Figure 4.8: Architecture of Linked Streaming Widgets

the server. Continuous Queries are registered at the server, using data from the data streams and static background knowledge to create results. These queries continuously generate result streams which are (i) sent to an endpoint, and (ii) can be reused in other continuous queries. Given the underlying architecture of C-SPARQL, this approach currently requires that data needs to be pulled from the data source and subsequently sent to the engine in a push-based manner. The engine currently does not provide support for stream processing where new data directly flows to the engine via, for instance, persistent HTTP connections. One barrier is that these persistent connections have to be supported and provided by the data sources which is rarely the case.

On the Linked Widgets Platform, we create so called Linked Streaming Widgets, which make use of the data served by the continuous queries. Depending on the implementation of these widgets, we can create different use cases covering and combining concepts such as sensor data based enrichment, aggregation, processing, and visualization. Linked Streaming widgets, hence, can facilitate the composition of sophisticated mashups which aim to incorporate stream data.

To initialize the C-SPARQL server (creating required streams and continuous queries, and serving static knowledge) we wrote a python wrapper. Relevant excerpts of the source code are included in Appendix D.

Linked Streaming Widgets are lightweight implementations which make use of the described architecture. They serve a specific purpose related stream data processing. For instance, they provide data for subsequent widgets, filter measurement data, or provide

means to monitor observations. The versatility of Linked Streaming Widgets is shown in the following use case descriptions.

4.3 Use Case: Citybike Mashup

The first use case deals with citybike data from the city of Vienna. We use a publicly available interface[12] to regularly retrieve the current state (free bikes, free boxes, available boxes) of the citybike network. We provide a collection of widgets which allows to analyze and visualize citybike usage over time to satisfy the individual information needs of the citizens. We follow the streaming widgets architecture and therefore develop (i) RML mappings to RDFize static and streaming knowledge, (ii) continuous queries operating over streamed and static RDF data, and (iii) widgets which utilize the results and can be integrated with other widgets.

The static knowledge is composed of information which is not expected to change frequently. Hence, we store information about each citybike station such as its geographic position (latitude, longitude), name, description, and identifier. The accompanying RML mapping (cf. Listing E.1 in the Appendix) is executed once and generates an RDF file which can then be used in continuous queries. An excerpt of the generated static knowledge is shown in Listing 4.3.

For the stream data we create a more complex RML mapping (cf. Listing E.2 in the Appendix) to match our environmental streams ontology created in Section 2. An excerpt of the resulting graph that represents a single observation is depicted in Figure 4.9.

```
<?xml version="1.0" encoding="UTF-8"?>
<rdf:RDF
   xmlns:geo="http://www.w3.org/2003/01/geo/wgs84_pos#"
   xmlns:qb="http://purl.org/linked-data/cube#"
   xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
   xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
   xmlns:ssn="http://purl.oclc.org/NET/ssnx/ssn#">

  <rdf:Description rdf:about="http://ldlab.ifs.tuwien.ac.at/envstreams/sensors/citybike/904">
    <geo:location rdf:resource="http://ldlab.ifs.tuwien.ac.at/envstreams/point/904"/>
    <rdf:type rdf:resource="http://ldlab.ifs.tuwien.ac.at/envstreams/ontology/Sensor"/>
    <rdfs:comment>bei Altem AKH vis a vis Otto-Wagner-Platz</rdfs:comment>
    <rdfs:label rdf:datatype="http://www.w3.org/2001/XMLSchema#string">Frankhplatz</rdfs:label>
  </rdf:Description>

  <rdf:Description rdf:about="http://ldlab.ifs.tuwien.ac.at/envstreams/point/904">
    <geo:long rdf:datatype="http://www.w3.org/2001/XMLSchema#string">16.355145</geo:long>
    <geo:lat rdf:datatype="http://www.w3.org/2001/XMLSchema#string">48.215156</geo:lat>
    <rdf:type rdf:resource="http://www.w3.org/2003/01/geo/wgs84_pos#Point"/>
  </rdf:Description>

</rdf:RDF>
```

Listing 4.3: Excerpt of static citybike station knowledge

[12] http://dynamisch.citybikewien.at/citybike_xml.php (accessed 16 June 2016)

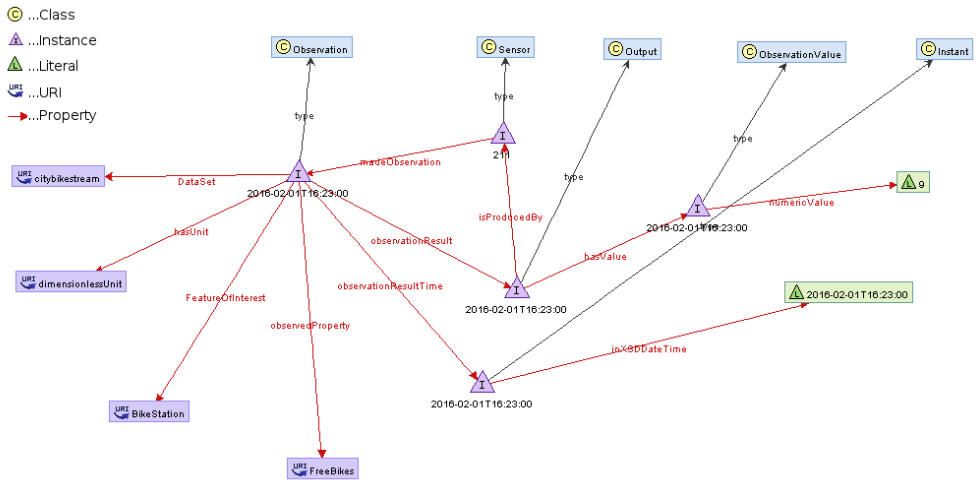


Figure 4.9: Excerpt of a graph showing a single observation of a sensor

These streams of RDF graphs are continuously created and fed into the C-SPARQL engine. Continuous queries are registered at the engine varying by window size and hence also varying by aggregation size. We create continuous queries that provide real time, hourly, daily, and weekly aggregates. To decrease the processing load, each query uses the output of the next smaller aggregate: The weekly aggregate query uses the results of the daily aggregate query as an input, the daily aggregate reuses the hourly and so on. An example query is shown in Listing F.1 in the Appendix. The query contains new keywords introduced by C-SPARQL [BBC+10b] that extend the SPARQL syntax: `REGISTER STREAM` is used to create a new stream. The URI which is defined at `REGISTER STREAM` can then be reused in other queries. `FROM STREAM` is used to denote the stream(s) the query should operate on. `RANGE` and `SLIDE` define the size and update frequency. The combination of `FROM STREAM` and `FROM` clauses in the same query enables the integration of stream data and static data. The presented query retrieves the latest citybike station measurements and returns them as an RDF graph.

In order to query for aggregated data, for instance to average observations over time, we need to register additional slightly adapted queries. A query computing a daily aggregate is constructed as follows (cf. Listing F.2 in the Appendix): The `CONSTRUCT` clause is similar to the real time query. In the `FROM STREAM` clause we need to define the output stream from the hourly aggregate as input stream used by the query. The window definition is adapted according to the desired type of aggregation, which, in our case is a daily aggregate. We define both the window size (`RANGE`) and slide (`SLIDE`) parameters as *24h*. To compute the actual aggregates, we define a subselect query which makes use of functions provided by SPARQL and Jena[13]. `afn:now()`, `SUBSTR()`, and `CONCAT()`

[13] Jena is a popular open source Java framework to build semantic web applications. C-SPARQL uses Jena. For more details visit `https://jena.apache.org/` (accessed 16 June 2016)

are used to create a new timestamp based on the results. `AVG()` computes the average and `IRI()` is used to create new URIs according to the chosen URI scheme. `STR()` and `xsd:dateTime()` is required for proper type conversion of retrieved values.

To complete the processing pipeline, the final step is to implement widgets, which reuse, combine, and integrate data provided by continuous queries. To this end, we developed the following widgets:

- Citybike Streaming: This widget is key for all use cases built upon citybike stream data. It is a data widget realized as a server widget providing as a data source to other widgets. It continuously collects citybike data from continuous queries and offers different aggregation levels to the user via its interface. Its output conforms to the RDF Data Cube vocabulary which is accepted by widgets, such as the generic *Map Viewer* widget.
- Citybike Streaming Storage: For monitoring use cases, where data is observed and stored over a longer period of time, we need a means to store data which is output by data widgets such as the *Citybike Streaming* widget. In the citybike use case, the *Citybike Streaming Storage* widget is responsible for this task. At its output terminal it provides data conforming to the RDF Data Cube vocabulary.
- Citybike Station Filter: This widget uses geographic locations, for instance served by the generic *Map Pointer* widget, at the input terminal to discover and filter for citybike stations based on *distance* or *number of results*. Users who are only interested in the data of specific stations should use this widget.
- Observation Filter: This widget is generic in that it accepts data based on the RDF Data Cube vocabulary which can then be filtered by *Measure Name*, *Operator* (greater, lesser, equal), and a *Threshold*. Hence, this widget only puts observations on the output, if the filter criteria match.
- Send Event: This widget can receive events at the input which are then sent to a person via mail or text message. This is useful for monitoring use cases in combination with the *Observation Filter* to receive an alert when specific values are below or above a defined threshold.

Furthermore, we reuse the already available *Map Pointer* and *Map Viewer* widgets. Due to their generic design and model, they can be also used in this use case.

Figure 4.10 shows a graphic representation of the semantic model for the Citybike Station Filter widget. We use the *lw:hasSampleData* property to associate the input and output terminals with descriptions that represent their full tree structures in JSON. As depicted, the widget accepts an instance of the type *dbp:Place* at the input, which is related to a *geo:Point* via the *geo:location* property. The *geo:Point* stores the latitude and longitude of the input point. The *lw:arrayDimension* value of *one* for *dbp:Place* reflects that the input is a list of points (1-dimensional array).

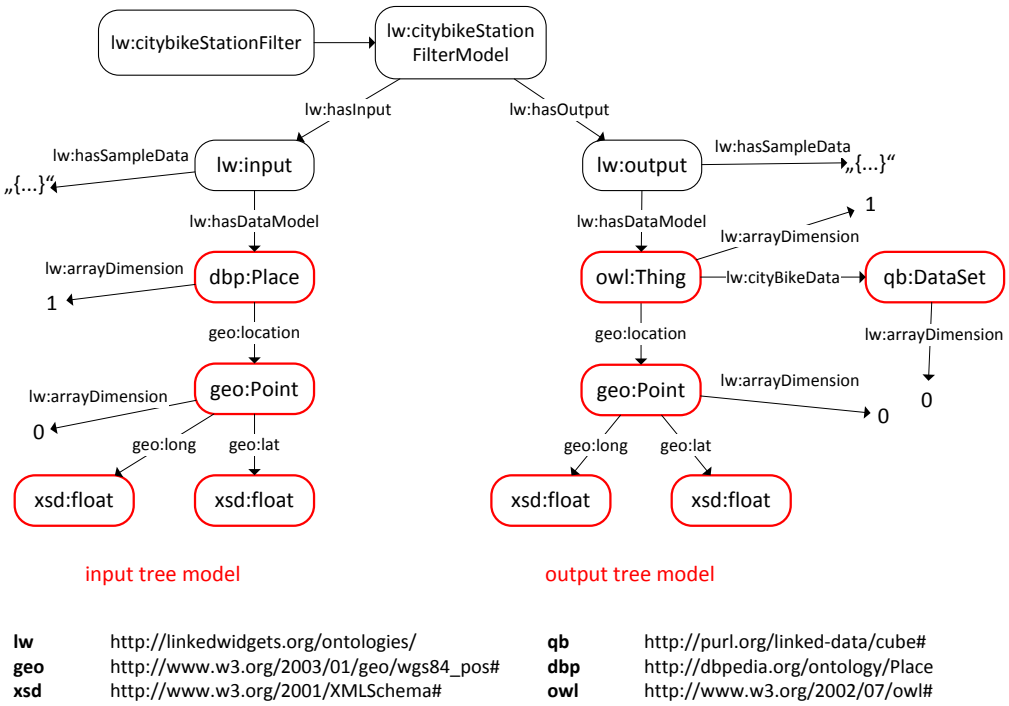


Figure 4.10: Semantic model of the Citybike Station Filter widget

The widget returns its output when it has completed its processing function. Again, we can see in the semantic model that the output can be an object of any type (*owl:Thing*). However this *owl:Thing* has to consist of a *geo:Point* and furthermore needs to include a *qb:DataSet* via the *lw:cityBikeData* relation. The semantic models of the *Citybike Streaming* and the *Citybike Streaming Storage* widgets are similar to the described model and hence omitted here. The models of the *Observation Filter* and *Send Event* widgets are shown in Appendix G.

Due to this modeling, we can reuse the *Map Pointer* to feed the *Citybike Station Filter* with a geographic location, which in turn is used to discover and filter citybike stations. Furthermore, since the widget's output includes a *qb:DataSet*, and the *Map Viewer* accepts this as an input we can use it to visualize the data.

Figures 4.11 and 4.12 show two mashups which can be composed with the developed widgets. Widgets with a green frame are server widgets, whereas a blue interface represents client widgets. In the first figure, we see a visualization of the real time citybike stream data, that is, *available bikes* and *available boxes* at the station which is selected via the filter widget. The user interface of the filter widget also allows to choose different types of temporal aggregates. The second figure represents a mashup which is useful in monitoring use cases. Again, a station is selected via the *Map Pointer* and *Citybike Station Filter* widget. Then the *Observation Filter* is used to define a threshold

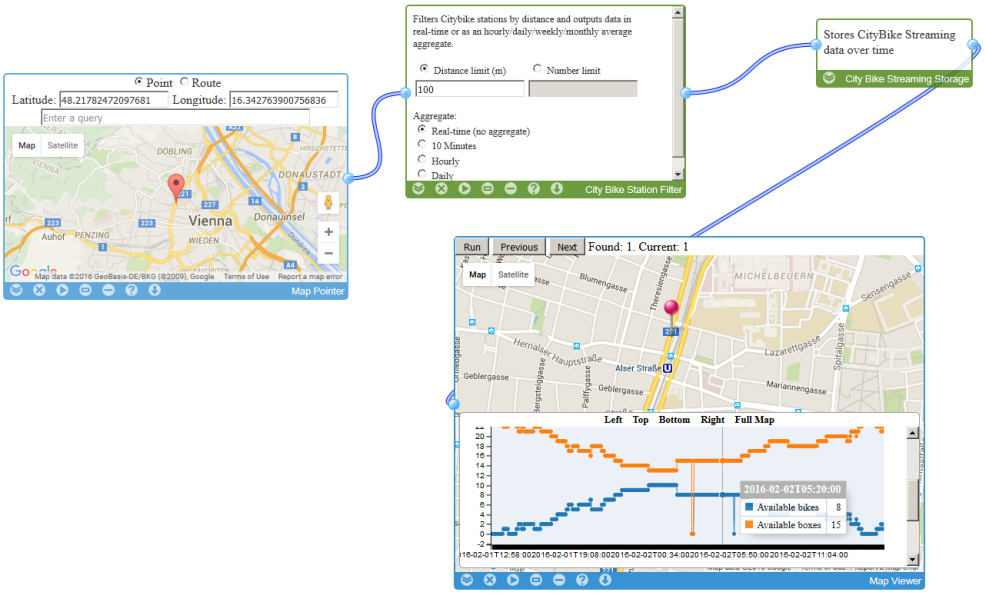


Figure 4.11: A mashup displaying citybike stream data of one specific station

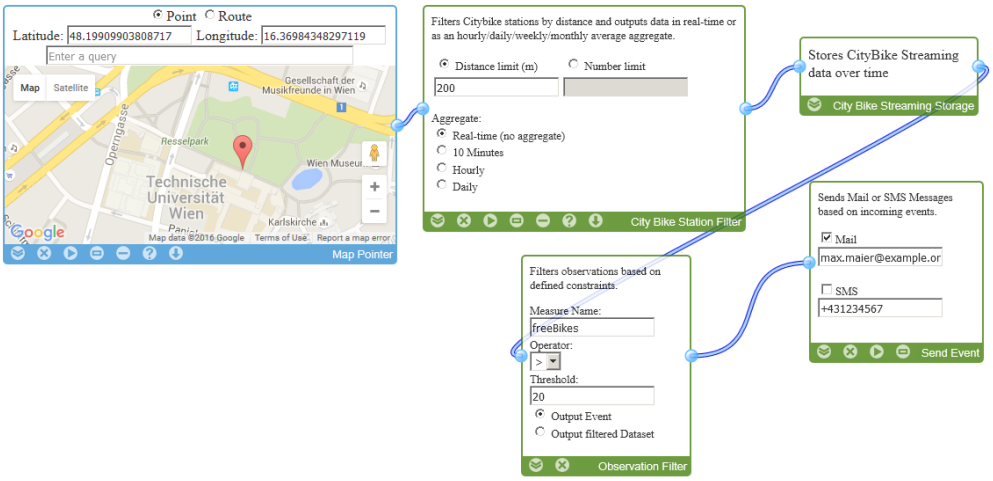


Figure 4.12: A mashup monitoring citybike observation data and triggering an event if a defined threshold is exceeded

based on a *Measure Name*, an *Operator*, and a *Threshold*. If the filter criterion is satisfied, an event is generated and sent to the *Send Event* widget, which will send the event as configured either via mail or text message.

To sum up, the citybike mashup use case acts as a proof of concept that with a limited amount of specially developed and carefully semantically modeled widgets, different types

of scenarios, such as data visualization or observation monitoring, can be realized. Correct modeling ensures that already available widgets can be reused, which in turn reduces the perceived complexity of users. Reusability is crucial in mashup environments, because it limits the amount of available widgets to a manageable number. The creation of a query network of aggregate queries reduces the processing load and amount of streamed triples on the server.

4.4 Use Case: Route Enrichment Mashup

As a second use case we show how stream data can be used to satisfy ad hoc information needs (cf. Figure 4.13). In this case a car driving route is enriched with real time air quality observation data so that citizens can choose routes based on air quality data. The data processing flow is similar to the citybike use case, however, we obviously use different RML mappings (cf. Appendix H) for both static (air quality sensors) and streaming (air quality observations) knowledge and different continuous queries (cf. Appendix I).

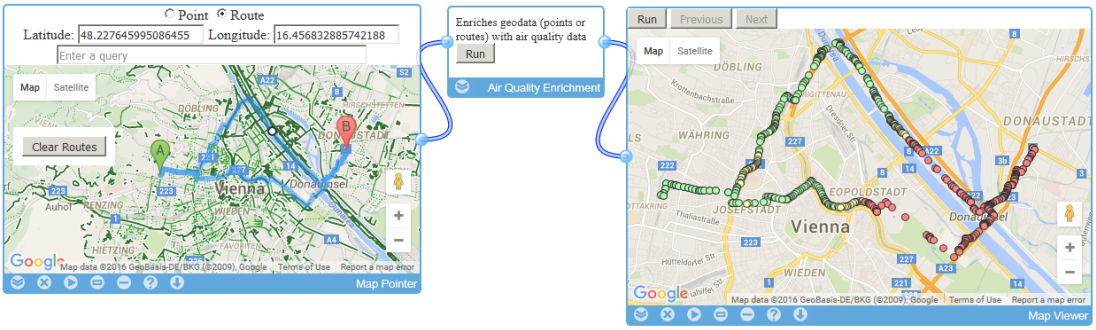


Figure 4.13: A mashup displaying routes which are enriched with stream data based on air quality observations

For the development of the use case, one new widget was developed and two other widgets were extended:

- Air Quality Enrichment: This widget receives a route as input. The output is again a route, but each point of the route is enriched with an air quality index which is calculated based on stream data observations. The semantic model of this widget is shown in Appendix J.
- Map Pointer: This widget needs to be able to output routes in order to be compatible with the *Air Quality Enrichment* widget. We therefore extended its semantic model accordingly, that is, adding routes to the output model.
- Map Viewer: The *Map Viewer* is also extended to be able to display routes which possibly are enriched with air quality data.

This use case showcases benefits which are gained with streaming widgets. Complementing the citybike scenario, this use case utilizes only client widgets to satisfy ad hoc information needs without requiring continuous monitoring of observation data. Instead, we use queries to integrate static knowledge (sensor metadata) and streaming data (air quality data) to calculate an air quality index. This index is then used to enrich route data and ultimately presented visually via the generic *Map Viewer* widget.

4.5 Summary of Use Cases

The presented use cases show that it is possible to implement an approach, i.e., "Linked Streaming Widgets", that facilitates the development of mashups which process environmental stream data. Users do not need to use programming languages, but they can use a visual editor to create their own solutions. We provide specific streaming widgets which process data that comes from specific sources and we develop generic widgets which use streaming data as an input, process it (e.g., filtering or storing the data), and make it available for subsequent widgets at the output.

The mashups which are shown in Figures 4.11–4.13 are just examples out of many other mashups that can be composed based on the available streaming widgets. The automatic mashup composition algorithm (cf. Section 4.2.1.6) is used to determine how many semantically and syntactically valid mashups can be composed for a given set of widgets. For the widgets provided in the city bike use case, 12 mashups can be composed, if the *Citybike Streaming* widget is used as an origin and 19 mashups can be composed, if the *Map Pointer* widget is the origin. Widgets provided in the route enrichment scenario can be used to compose 9 different types of mashups, if the *Map Pointer* widget is the starting point. This shows that already a small number of widgets enable the composition of many different mashup solutions. However, providing new streaming data sources as widgets is crucial to facilitate the creation of widgets in new application domains and to foster further ways of integration.

4.6 Evaluation

We quantify performance characteristics of the Linked Streaming Widgets approach in order to show its practical and technical feasibility. Further, we evaluate and compare two different approaches to develop and process the required queries. To this end, we conduct a thorough analysis of query processing performance at the C-SPARQL engine. Namely, we measure *query execution time*, *triples per stream over time*, and *memory consumption over time* for two different scenarios based on the citybike use case (cf. Section 4.3).

The two analyzed approaches are subsequently called *Construct* and *Select* approach due to their reliance on either `CONSTRUCT` or `SELECT` queries. In order to realize a use case which also requires to compute aggregate results, we can choose between two alternatives:

- *Select approach*: We push the real time data into the engine on a single stream and then register separate queries for each type of aggregate we want to calculate. The source stream for each query (integrated via the `FROM STREAM` clause) is the same for each query, namely the stream which the real time data is put on.
- *Construct approach:* In this case, we use one data stream to push the source data into the engine. However, the real time stream is only used by one query which computes the results based on this data. In addition, we create a cascade of `CONSTRUCT` queries. These queries allow to stream their results into a new and separate stream which can then be consumed by another query (cf. Appendix K).

Both approaches differ in that the *Select* approach provides more precise answers, because the queries are executed over the complete sets of measurements in a window. In contrast, the *Construct* approach executes queries over aggregated data which may result in inaccurate results, but will likely yield better performance. Since the `CONSTRUCT` queries are cascaded, the approach also introduces interdependencies between the queries. For instance, if the second query in the cascade – for some reason – stops working, then all subsequent queries will also stop working, because they are dependent from the results of the second query. This behavior does not occur for the `SELECT` queries.

In the following sections we present the results of our evaluation comparing the performance of both approaches along different dimensions. We conducted experiments for different aggregate queries: (i) 2 minute real time, (ii) 10 minute aggregate, (iii) 1 hour aggregate, and (iv) 6 hour aggregate. The 6 hour aggregate ran successfully only for the *Construct* approach. We suspect that for the *Select* approach there were too many triples on the stream after six hours of runtime, which caused the engine to fail while trying to computing the results.

The experiments were conducted on an Intel Xeon CPU E5-2620 0 @ 2.00GHz, Hexa Core, 4 GB RAM running SMP Debian 3.2.54-2 and ran for 24 hours. We used the REST version of C-SPARQL called rsp-services-csparql 0.4.9 available at GitHub[14].

4.6.1 Results

The results of the evaluation are:

- **Query execution time evaluation**: Figures 4.14a–c show the results of this experiment for different aggregate queries varying in window size. The window size also determines how often the query is executed. The figures show how long the execution of the queries took over the course of the experiment.
- **Triples on stream**: Figures 4.15a–c show the amount of triples on the stream while running the experiment for varying aggregate queries.

[14] `https://github.com/streamreasoning/rsp-services-csparql` (accessed 16 June 2016)

- **Memory usage**: Figure 4.16 shows the memory consumed over time by the stream processing engine for both approaches.

Based on the results we can draw a number of conclusions.

4.6.1.1 Query execution time

C-SPARQL executes continuous queries every time a window closes. At this time, the engine considers all triples within the window scope and computes the results. Hence, bigger windows contain more triples and therefore yield longer execution times. In our evaluation we observe that in the case of 2 minute windows, query execution time is on average five times lower for the *Construct* approach (average = 40ms) than for the *Select* approach (average = 190ms). This gap even increases when we look at the bigger window sizes of 10 minutes (*Construct* 36 times faster) and 1 hour (*Construct* 24 times faster). For 6 hour windows, the *Select* approach failed, hence, we only show the results for the *Construct* approach in Figure 4.14c. Generally, the execution time of the *Construct* queries is good even for big windows where execution time exceeds 1sec slightly. However, for *Select* queries the execution times are not acceptable for such window sizes (average = 240sec).

4.6.1.2 Triples on stream

The gaps between the performances of both approaches can be explained by the amount of triples available on the stream which a query uses to compute its results. The use of `CONSTRUCT` queries limits the amount of triples on the streams which are used by the queries, while the `SELECT` queries always operate over the complete real time stream. The results of this experiment quantify this gap, showing that the difference of triples between both approaches increase dramatically for bigger windows. For 10 minute windows there are five times as many triples stored on a stream for the *Select* approach, for 1 hour windows there are four times as many. Obviously, it is more resource-intensive for an engine to compute a query which is executed over five times as many triples.

4.6.1.3 Memory consumption

The memory consumption results show that in general the *Construct* approach is more memory efficient. For both scenarios memory usage oscillates, because triples are deleted when a query for a closing window is computed and these triples are not part of another additional window. However, we also observe that, even though triples are deleted after query executions, memory consumption steadily increases over the course of the experiment. This suggests that the engine fails to delete outdated triples which leads to unnecessarily high memory consumption over time. The sudden decline at the 800

minute mark at the *Construct* approach depicts a forced garbage collection by the Java Virtual Machine[15].

4.6.1.4 Summary

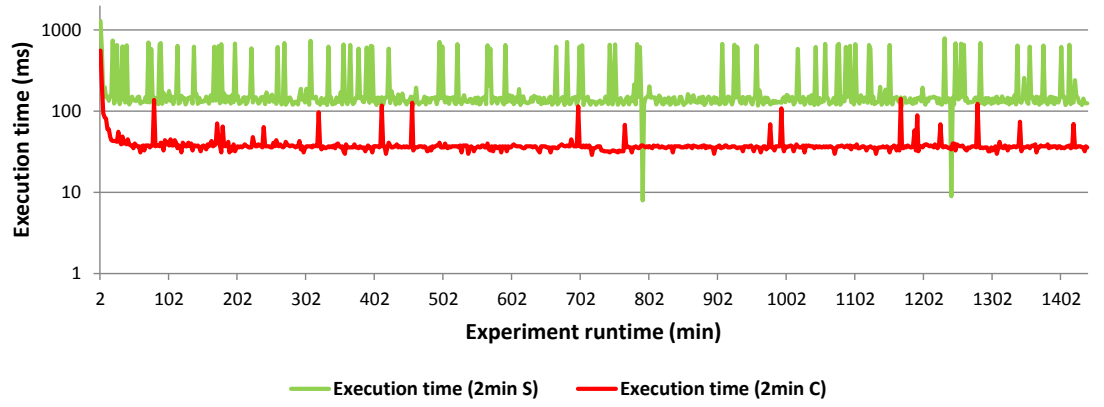
The results show that the *Construct* approach yields better performance with respect to query execution time, triples per stream, and memory consumption. The main reason for this is the reduced amount of triples pushed to the streams which need to be processed by the queries. This performance increase also influenced our decision on implementing a `CONSTRUCT` network of continuous queries when we created the widgets for the citybike use case (cf. Section 4.3). The approach demonstrates the feasibility of the technical implementation of the *Linked Streaming Widgets* concept for the given use cases.

Work on optimization of data and query processing is still scarce, because the research field of semantic stream processing is still in its infancy. We show how the use of query networks outperforms the alternative approach which uses `SELECT` queries. To guarantee more scalability, for instance in distributed environments, further research is necessary. As an initital step, CQELS Cloud has been presented, which extends CQELS towards parallel processing in the cloud [LPQLVH13]. The authors of CQELS Cloud show increased performance and scalability of continuous query operators running on the Amazon EC2 platform. Hoeksema and Kotoulas [HK] present an extension of C-SPARQL for the S4 streaming platform also showing favorable throughput improvements.

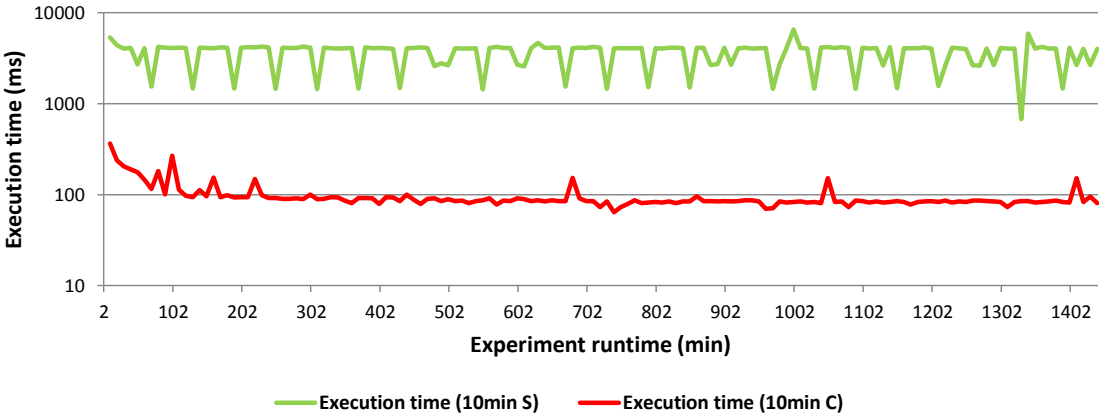
Another research direction to optimize query processing in a stream environment aims at reducing the cost of expensive updates for static knowledge which often is integrated in continuous queries. The infrequently changing data of the static knowledge usually is retrieved each time a query is computed. Given that updates on this data rarely happen, valuable processing time may be saved by avoiding or minimizing these updates. In [DDG+15, DMD+15] Dehghanzadeh et al. present first promising results of their approach called *Window Based Maintenance*.

Further paths of research to increase the performance of stream processing approaches are *approximate reasoning* [RPZ10], *incremental reasoning* [BBC+10a, CSGL15], or reasoning with Graphics Processing Units (GPU)s [LUQ14].

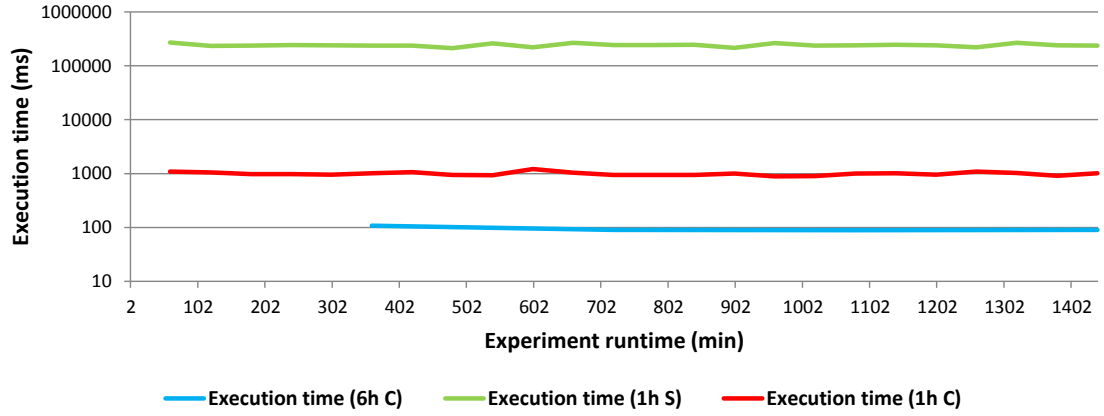
[15] We ran C-SPARQL with the following command: `java -XX:-UseConcMarkSweepGC -Xms2g -Xmx2g -jar [Path to JAR]`



(a) Query execution time for the 2 minute queries

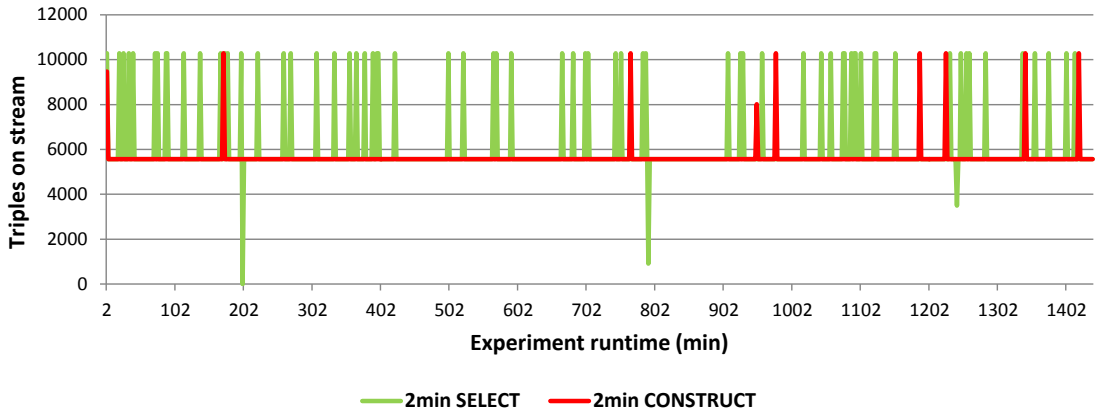


(b) Query execution time for the 10 minute aggregate queries

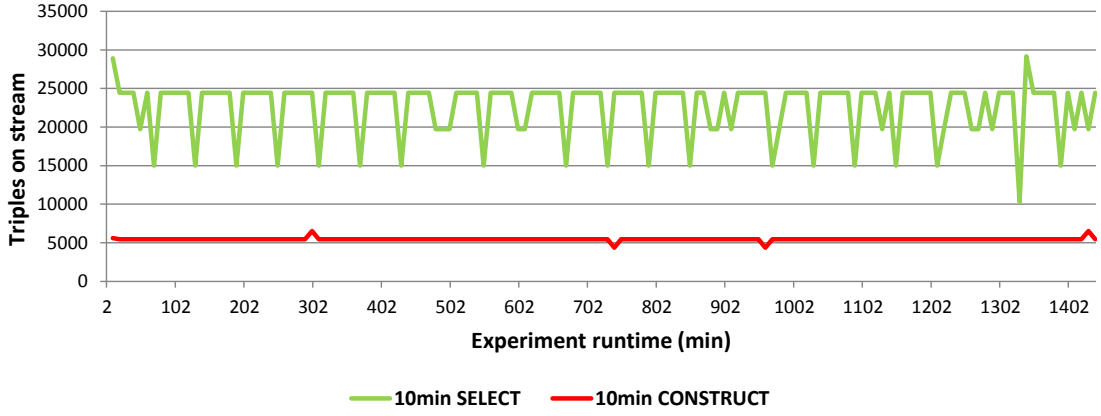


(c) Query execution time for the 1 hour and 6 hour aggregate queries

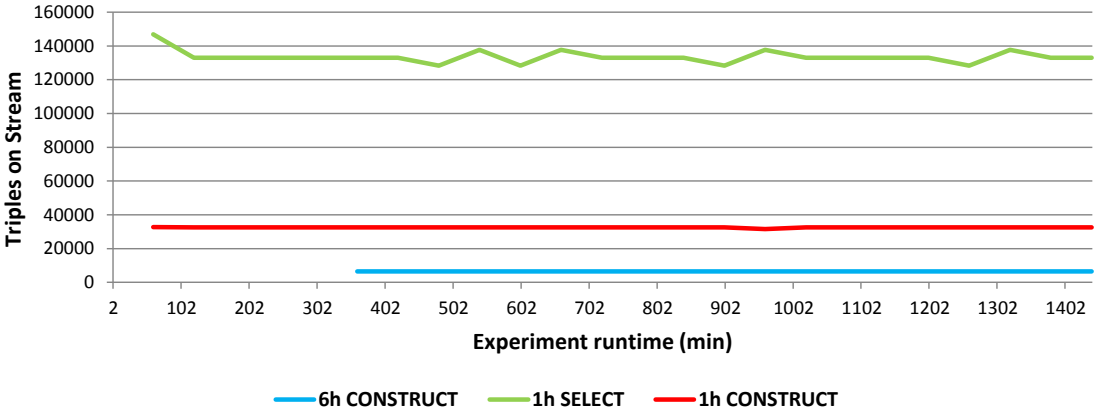
Figure 4.14: Query execution time evaluation results for both *Select* (S) and *Construct* (C) approaches.



(a) Triples on stream over time for the 2 minute queries



(b) Triples on stream over time for the 10 minute aggregate queries



(c) Triples on stream over time for the 1 hour and 6 hour aggregate queries

Figure 4.15: Triples on stream evaluation results for both *Select* (S) and *Construct* (C) approaches.

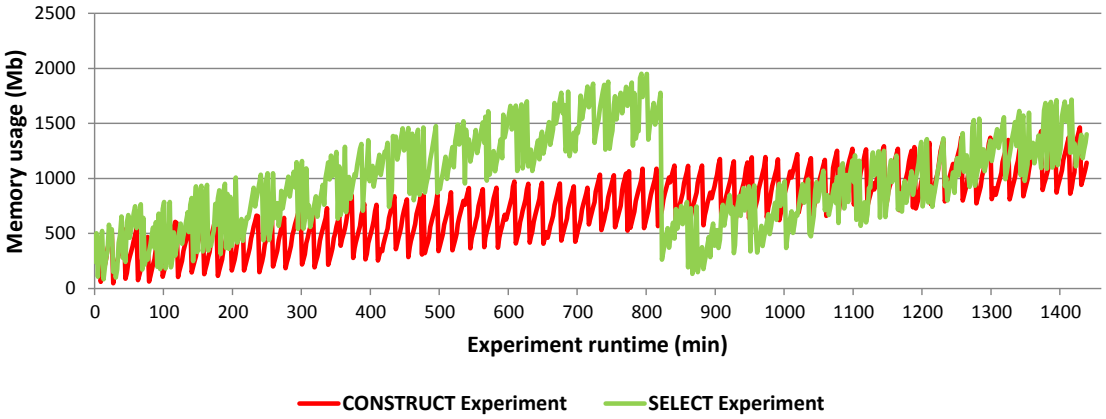


Figure 4.16: Memory usage over time by the stream processing engine C-SPARQL for both the *Select* and *Construct* approaches

Part III

Conclusion

CHAPTER 5

Summary

The state of our world's environment is critical. Global organizations such as the United Nations Organization (UNO) [UNE16], European Environment Agency (EEA) [EEA15], and the World Health Organization (WHO) [WHO16] frequently raise awareness for issues like global warming, increasing air/water/soil pollution, resource inefficiency, among others. Cities have often been criticized, because they cause environmental issues due to increased waste, traffic, etc. Recently they are also considered as offering huge potential to face anthropogenic environmental challenges [GFG+08]. In addition, the popular term *Smart City* describes the concept of an intelligent city based on the integration of technological solutions. The vision of a Smart City is to cope with and solve major environmental challenges; this has generated a lot of interest in both industrial and academic spheres and has, for example, been strongly supported by EU initiatives.

Ubiquitously available sensors generate timely data about our environment, potentially supporting city stakeholders, such as citizens, urban planners, and politicians. However, there is a lack of readily available applications making use of and contextualizing the amounts of data which is generated continuously. Exploiting the full potential of this data would improve informed decision-making and could ultimately lead to a more eco-friendly way of living. Examples are reduction of individual traffic, optimizing public transport, intelligent waste management, and smart grids to improve the efficiency of electricity operations.

The work of the present thesis deals with the challenges of exploiting environmental data created in urban areas to improve decision making for city stakeholders.

The first part gives a motivation and description of current problems in the area of environmental data integration. We identify a central research question, namely "*Can real time environmental data be provided to create actionable knowledge for urban stakeholders?*". This question subsequently is divided into three subquestions dealing with (i) the integration of heterogeneous environmental stream data sources, (ii) the evaluation

of semantic stream processing systems, and (iii) the utilization of environmental data streams by city stakeholders. Further, we report on the used methodology, main contributions, structure of the work, and publications which were written over the course of this research.

The second part deals with our approach on how to provide data streams to end users. First, we conduct a requirements analysis and review existing ontologies to formulate a new vocabulary which can be used to model environmental data streams. This *Environmental Streams Vocabulary* extends and reuses concepts of the de-facto standard ontologies SSN and RDF Data Cube. It can be used to model data streams capturing environmental data while being compliant with the RDF Data Cube vocabulary and the Semantic Sensor Network ontology. This facilitates reuse and data integration with sources which already make use of these ontologies.

Second, we provide a methodology to benchmark semantic stream processing engines (also known as RSP engines) called *YABench*. To the best of our knowledge YABench is the first framework that enables to compare RSP engines with respect to correctness of results and performance on a granular level. We use this framework to assess which engine is most suitable with respect to the requirements that are deduced from the challenge of providing environmental data streams to end users. Our experiments show that when comparing C-SPARQL to CQELS, the former produces shorter delays in result delivery when running simulations for different amounts of sensor stations while providing high precision and recall. Similar behavior is also found when we perform experiments with more complex queries. C-SPARQL shows better performance in terms of memory consumption and finally, YABench is able to reveal erroneous behavior of CQELS. These findings support the conclusion that C-SPARQL is the preferred engine to provide environmental data streams to end users.

Finally, we extend the Linked Widgets Platform with the concept of *Linked Streaming Widgets*. Linked Streaming Widgets allow end users to build light-weight web applications facilitating integration and reuse of environmental data stream sources. Users are provided with a visual interface which abstracts away the complexity of stream processing and data integration. The platform executes data acquisition, transformation, processing, and registration of queries in the background and enables users to compose applications. We demonstrate the feasibility of this approach by means of two use cases dealing with citybike data and integration of traffic route data with air quality data. We compare two implementation approaches of the processing architecture and evaluate to which extent the construction of query networks outperforms the traditional approach of using queries which do not depend on each other.

On a general level the contributions of the present thesis have several implications: It is evident that stream processing of heterogeneous data, such as environmental data, necessitates the development of ontologies which have characteristics that are aligned with the nature of the data. In order to build such an ontology a sound requirements analysis which takes into account characteristics of the data, such as its volume, frequency, and variety, is required. Still, more research needs to be done on how to design ontologies

which can be used in a stream setting and allow for scalable reasoning. The evaluation of RSP engines presented in this work has shown that currently engines are either not scalable enough to deal with streams of moderate frequency, or show incorrect behavior in some cases. Hence, advancements towards increasing the performance and correctness of RSP engines to deal with realistic settings of high-frequency and high-volume data streams are still required. Another lesson we learned deals with the provision of user friendly interfaces to enable the integration of web data with data streams for non-expert users who lack programming skills. The design of such an interface requires a trade-off between *usability* and *functionality*. The more complexity is added to the user interface, the more functionality the user will gain, but the system will be more difficult to use. This trade-off needs to be carefully considered taking into account the target users and their information needs.

CHAPTER 6

Answers to Research Questions

We answer the research questions stated in Section 1.3 as follows:

R1 *What data integration methods can be used to model environmental real time data to overcome heterogeneity and to allow reusability and explorability?*

In Chapter 2 we define a set of requirements and show that the use of data modeling capabilities satisfy them. More precisely, we use controlled vocabularies, i.e., ontologies, to provide a controlled schema for semantically describing environmental sensor data streams supporting integration of novel and reuse of existing data sources.

R2 *How can we evaluate systems for semantic processing of real time environmental data streams?*

The *YABench* framework presented in Chapter 3 is the basis to identify a suitable semantic stream processing system based on the requirements imposed by environmental stream data. The modular framework allows to define environmental stream scenarios and collects quantifiable results based on granular metrics. These metrics include hardware consumption and correctness evaluations. This supports the decision on which system fits best for a given scenario, i.e., environmental stream data in our case.

R3 *How can non-expert users, i.e, urban decision makers, be enabled to explore environmental stream data?*

The concept of *Linked Streaming Widgets* and the implementation of two example use cases introduced in Chapter 4 show the feasibility and applicability of the approach to support non-expert users in making improved decisions based on

environmental stream data. The integration of both static and real time streaming knowledge contextualizes the provided information and therefore further raises its usefulness. *Linked Streaming Widgets* hide the complexity of stream processing from users and fulfill the requirements of the end user programming paradigm.

To sum up, based on presented contributions the central research question

Can real time environmental data be provided to create actionable knowledge for urban stakeholders?

can be answered positively.

CHAPTER 7

Future Work

The evaluation of the implementation of our methods shows that it is possible to provide users with environmental data and to support real time decision making based on data integration methods. The work of this thesis can be extended in several directions:

Archiving Data Streams. This work presents an approach to explore and utilize real time environmental data. However, for deeper analysis, such as detecting patterns, stakeholders may also be interested in historic data, that is, archives of the data streams. Currently, semantic stream processing engines do not enable access to expired data items which moved out of a window's scope. The development of techniques to archive data streams and make them available ex post is a non trivial task. Just storing all expiring data items in a triple store will not be scalable for scenarios such as environmental sensor data. To this end, efficient solutions to compress, serialize, and deserialize big RDF data are required. Initial approaches have already been proposed [UMD+13, GFM15], however, their interrelation with stream processing systems have not been tackled, yet.

Open Data Quality. In line with the provision of real time open data, the quality of released data sets, especially of time-dependent data, is crucial. Data providers often publish data sets without taking into account how the data may be used or combined with already available data. This shortcoming decreases the quality of open data portals and exacerbates (re)use of the data. Initial work on the assessment of open data and open data portals is already under way [UNP15, BETL12]. Still, we need more insights on how a high quality level of open data, particularly real time data, can be provided to foster use and recombination of the data. Moreover, completeness and correctness are also important factors which influence the overall quality of published data sets and need to be considered by the providers.

Transformation of Streams to Linked Data. There is a need to ease the transformation of raw data to linked data for potential developers in order to facilitate linked data publication. In our approach we use RML mappings to perform this conversion on-the-fly. The mappings are to be defined only once, however, this process is still cumbersome and requires deep knowledge of the underlying data structures. *TripleWave* is an initial proposal to provide a system that allows to deploy RDF streams on the web [MCD+15]. Still, research into efficient ways of transforming raw data on-the-fly into linked data streams is needed to realize a *Web of Data Streams*.

Benchmark Extensions. There are several directions for future research to extend the proposed benchmark.

First, the functional coverage of the test cases can be increased. It would be interesting to evaluate the influence of multiple windows in one query on an engine. To this end, it would be necessary to extend the oracle to support other window operators and combinations of multiple windows.

Second, the integration of big background knowledge and multiple input streams would be a valuable asset. This will broaden our understanding of how well engines can deal with merging high-frequency data streams with large static data sources, which is one of the promising application scenarios for RSP engines.

Third, the identification of further insights on how engines cope with different real-world streaming scenarios necessitates the implementation of different types of data generators. For instance, in a social media scenario, data is likely to arrive in bursts. In such settings, engines need to be able to deal with strong variations in inter-arrival times of elements and being able to quantify their behavior under such circumstance will be revealing.

Fourth, complex reasoning beyond simple RDFS entailment is not yet covered by YABench, nor by any other existing benchmark. This will become relevant once it is fully supported by available engines.

Last, the implementation of benchmarks allowing for the registration of multiple queries at the same time will cover a broader set of realistic use cases.

Decoupling Streaming Extensions. The *Linked Streaming Widgets* approach presented in this thesis has shown that it is feasible to build applications on top of RSP engines, in our case C-SPARQL. Because there are currently several proposals for RSP engines and that it is therefore unlikely that a single standardized engine will be adopted, it is crucial that such applications are decoupled from underlying RSP architectures. This decoupling will facilitate the processing of external streams and ease the integration of additional modules in the pipeline. To this end, it will be necessary to conduct research towards decoupled event processing systems to enable the design of RSP applications irrespective of the underlying engines. Some early ideas to solve this challenge exist, such as unifying RSP semantics [DCVC15],

sharing and reusing continuous queries [KB15b], and decoupling event processing systems [KB15a, CA15].

Streaming Widgets Flexibility. The presented use cases of streaming widgets demonstrate the power of combining data streams with mashup development. However, there are still many possibilities to increase the user experience. Ways that enable to work with multiple data streams, parametrize queries, and register new queries have to be investigated. On a more general level, users should be able to aggregate data more easily and to define more complex patterns for matching in monitoring use cases. This would greatly enhance the versatility of the approach and of the platform.

Appendix

A Experiment 1 queries

```
REGISTER QUERY test AS

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

SELECT ?sensor ?obs ?value
FROM STREAM <http://ex.org/streams/test> [RANGE ${WSIZE} STEP ↩
   ${WSLIDE}]
WHERE {
  ?obs om-owl:observedProperty weather:_AirTemperature ;
       om-owl:procedure ?sensor ;
       om-owl:result [om-owl:floatValue ?value] .
  FILTER(?value > ${TEMP})
}
```

Listing A.1: C-SPARQL query for YABench experiment one

```
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 ?obs ?value WHERE {
  STREAM <http://ex.org/streams/test> [RANGE ${WSIZE} SLIDE ↩
     ${WSLIDE}] {
    ?obs    om-owl:observedProperty weather:_AirTemperature ;
                om-owl:procedure ?sensor ;
                om-owl:result ?res .
    ?res om-owl:floatValue ?value .
  }
  FILTER(?value > ${TEMP})
}
```

Listing A.2: CQELS query for YABench experiment one

B Experiment 2 queries

```
REGISTER QUERY test AS

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

SELECT (AVG(?value) AS ?avg)
FROM STREAM <http://ex.org/streams/test> [RANGE ${WSIZE} STEP ↩
   ${WSLIDE}]
WHERE {
  ?obs om-owl:observedProperty weather:_AirTemperature ;
       om-owl:procedure ?sensor ;
       om-owl:result [om-owl:floatValue ?value] .
       FILTER(?value > ${TEMP})
}
```

Listing B.1: C-SPARQL query for YABench experiment two

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

SELECT (AVG(?value) AS ?avg) WHERE {
  STREAM <http://ex.org/streams/test> [RANGE ${WSIZE} SLIDE ↩
     ${WSLIDE}] {
    ?obs    om-owl:observedProperty weather:_AirTemperature ;
            om-owl:procedure ?sensor ;
            om-owl:result ?res .
    ?res om-owl:floatValue ?value .
  }
  FILTER(?value > ${TEMP})
}
```

Listing B.2: CQELS query for YABench experiment two

C Experiment 3 queries

```
REGISTER QUERY test AS

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

SELECT ?sensor ?ob1 ?value1 ?ob2
FROM STREAM <http://ex.org/streams/test> [RANGE ${WSIZE} STEP ↩
    ${WSLIDE}]
WHERE {
  ?ob1 om-owl:procedure ?sensor ;
       om-owl:observedProperty weather:_AirTemperature ;
       om-owl:result ?res1 .
  ?res1 om-owl:floatValue ?value1 .
  ?ob2 om-owl:procedure ?sensor ;
       om-owl:observedProperty weather:_AirTemperature ;
       om-owl:result ?res2 .
  ?res2 om-owl:floatValue ?value2 .
  FILTER(?value1-?value2 > ${VARIATION_THRESHOLD})
}
```

Listing C.1: C-SPARQL query for YABench experiment three

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

SELECT ?sensor ?ob1 ?value1 ?ob2
WHERE {
  STREAM <http://ex.org/streams/test> [RANGE ${WSIZE}] {
    ?ob1 om-owl:procedure ?sensor ;
       om-owl:observedProperty weather:_AirTemperature ;
       om-owl:result ?res1 .
    ?res1 om-owl:floatValue ?value1 .
    ?ob2 om-owl:procedure ?sensor ;
       om-owl:observedProperty weather:_AirTemperature ;
       om-owl:result ?res2 .
    ?res2 om-owl:floatValue ?value2 .
  }
  FILTER(?value1-?value2 > ${VARIATION_THRESHOLD})
}
```

Listing C.2: CQELS query for YABench experiment three

D Python wrapper for C-SPARQL

```
import requests
import settings
import time
import urllib.request
from rdflib import Graph
import log
import urllib.parse
import sys

def registerStream(streamuri):
    try:
        r = requests.put("http://{}/streams/{}".format( ↩
            settings.CSPARQL_SERVERURL, ↩
            urllib.parse.quote_plus(streamuri)))
        logger.debug("{} - at ↩
            {}".format(r.text,settings.CSPARQL_SERVERURL))
    except requests.exceptions.RequestException as e:
        logger.debug(e)
        sys.exit(1)

def putStaticModel(staticKB):
    try:
        #read static knowledge and serialize to turtle
        response = urllib.request.urlopen(staticKB)
        data = response.read()
        g = Graph()
        try:
            g.parse(data=data, format="xml")
        except Exception as e:
            logger.debug(e)
            sys.exit(1)
        turtle_string = g.serialize(format='turtle', ↩
            encoding='utf-8',).decode("utf-8").replace('\n',' ')

        r = requests.post("http://{}/kb".format( ↩
            settings.CSPARQL_SERVERURL), data={"action":"put", ↩
            "iri" : staticKB, "serialization" : turtle_string})
        logger.debug("{} - at ↩
            {}".format(r.text,settings.CSPARQL_SERVERURL))
    except requests.exceptions.RequestException as e:
        logger.debug(e)
        sys.exit(1)

def registerQuery(queryname,queryobserver):
    #register query
```

```
    try:
        r = requests.put("http://{}/queries/{}".format( ↩
            settings.CSPARQL_SERVERURL, ↩
            queryname),data=settings.QUERYDICT[queryname])
        logger.debug("{} - at ↩
            {}".format(r.text,settings.CSPARQL_SERVERURL))
    except requests.exceptions.RequestException as e:
        logger.debug(e)
        sys.exit(1)

    #add observer
    try:
        r = requests.post("http://{}/queries/{}".format( ↩
            settings.CSPARQL_SERVERURL, ↩
            queryname),data=queryobserver)
        logger.debug("{} - {} {}".format(r.text,'observer added ↩
            at', queryobserver))
    except requests.exceptions.RequestException as e:
        logger.debug(e)
        sys.exit(1)


if __name__ == '__main__':
    logger = log.setup_custom_logger('root')
    putStaticModel(settings.AIRQUALITY_STATICURI)
    registerStream(settings.AIRQUALITY_STREAMURI)
    registerQuery(settings.AIRQUALITY_QUERYNAME, ↩
       settings.AIRQUALITY_QUERY_DESTINATION)
```

Listing D.1: Python source code calling REST functions of the C-SPARQL server

E RML mappings for the citybike use case

```
@prefix rml: <http://semweb.mmlab.be/ns/rml#>.
@prefix ql: <http://semweb.mmlab.be/ns/ql#>.
@prefix r2rml: <http://www.w3.org/ns/r2rml#>.
@prefix wgs: <http://www.w3.org/2003/01/geo/wgs84_pos#>.
@prefix ont: <http://ldlab.ifs.tuwien.ac.at/envstreams/ontology/>.

<Mapping1>
    rml:logicalSource
            [ rml:iterator
               "//station";
            rml:referenceFormulation
               ql:XPath;
            rml:source
               "http://dynamisch.citybikewien.at/citybike_xml.php" ];
    r2rml:predicateObjectMap
            [ r2rml:objectMap
               [ rml:reference "name" ;
                 r2rml:datatype xsd:string ];
            r2rml:predicate
               <rdfs:label> ],
            [ r2rml:objectMap
                    [ r2rml:template
                       "http://ldlab.ifs.tuwien.ac.at/envstreams/point/{id}";
                    r2rml:termType
                       r2rml:IRI ];
            r2rml:predicate
               wgs:location ],
            [ r2rml:objectMap
               [ rml:reference "description" ];
            r2rml:predicate
               <rdfs:comment> ];
    r2rml:subjectMap
            [ r2rml:class
               ont:Sensor;
            r2rml:template
               "http://ldlab.ifs.tuwien.ac.at/envstreams/sensors/citybike/{id}" ].
<Mapping2>
    rml:logicalSource
            [ rml:iterator
               "//station";
            rml:referenceFormulation
               ql:XPath;
            rml:source
               "http://dynamisch.citybikewien.at/citybike_xml.php" ];
    r2rml:predicateObjectMap
            [ r2rml:objectMap
               [ rml:reference "longitude" ;
                 r2rml:datatype xsd:string ];
            r2rml:predicate
               wgs:long ],
            [ r2rml:objectMap
               [ rml:reference "latitude" ;
                 r2rml:datatype xsd:string ];
            r2rml:predicate
               wgs:lat ];
    r2rml:subjectMap
            [ r2rml:class
               wgs:Point;
            r2rml:template
               "http://ldlab.ifs.tuwien.ac.at/envstreams/point/{id}" ].
```

Listing E.1: RML mapping to create static knowledge for the citybike use case

```
@prefix rml: <http://pebbie.org/mashup/rml-source/>.
@prefix rm: <http://semweb.mmlab.be/ns/rml#>.
@prefix ql: <http://semweb.mmlab.be/ns/ql#>.
@prefix r2rml: <http://www.w3.org/ns/r2rml#>.
@prefix ssn: <http://purl.oclc.org/NET/ssnx/ssn#>.
@prefix ont: <http://ldlab.ifs.tuwien.ac.at/envstreams/ontology/>.
@prefix xsd: <http://www.w3.org/2001/XMLSchema#>.
@prefix rel: <http://sweet.jpl.nasa.gov/2.3/relaSci.owl#>.
@prefix cube: <http://purl.org/linked-data/cube#>.
@prefix qudt: <http://qudt.org/schema/qudt#>.
@prefix time: <http://www.w3.org/2006/time#>.

rml:freebikes
    rm:logicalSource
            [ rm:iterator
               "//station";
            rm:referenceFormulation
               ql:XPath;
            rm:source
               "http://dynamisch.citybikewien.at/citybike_xml.php" ];
    r2rml:predicateObjectMap
            [ r2rml:objectMap
                    [ r2rml:template
                        "http://ldlab.ifs.tuwien.ac.at/envstreams/sensors/citybike/{id}/↩
                             freebikesobservation/{TIMESTAMP}";
                    r2rml:termType
                       r2rml:IRI ];
            r2rml:predicate
               ssn:madeObservation ];
    r2rml:subjectMap
            [ r2rml:class
               ont:Sensor;
            r2rml:template
               "http://ldlab.ifs.tuwien.ac.at/envstreams/sensors/citybike/{id}" ].
rml:observationfreebikes
    rm:logicalSource
            [ rm:iterator
               "//station";
            rm:referenceFormulation
               ql:XPath;
            rm:source
               "http://dynamisch.citybikewien.at/citybike_xml.php" ];
    r2rml:predicateObjectMap
            [ r2rml:objectMap
                    [ r2rml:constant
                        "http://sweet.jpl.nasa.gov/2.3/reprSciUnits.owl#dimensionlessUnit";
                    r2rml:termType
                       r2rml:IRI ];
            r2rml:predicate
               rel:hasUnit ],
            [ r2rml:objectMap
                    [ r2rml:constant
                       "http://ldlab.ifs.tuwien.ac.at/envstreams/citybikestream";
                    r2rml:termType
                       r2rml:IRI ];
            r2rml:predicate
               cube:DataSet ],
            [ r2rml:objectMap
                    [ r2rml:template
                        "http://ldlab.ifs.tuwien.ac.at/envstreams/sensors/citybike/{id}/↩
                             freebikesobservation/output/{TIMESTAMP}";
                    r2rml:termType
                       r2rml:IRI ];
            r2rml:predicate
               ssn:observationResult ],
            [ r2rml:objectMap
                    [ r2rml:constant
                        "http://ldlab.ifs.tuwien.ac.at/envstreams/ontology/FreeBikes";
                    r2rml:termType
                       r2rml:IRI ];
            r2rml:predicate
               ssn:observedProperty ],
```

```
            [ r2rml:objectMap
                    [ r2rml:template
                        "http://ldlab.ifs.tuwien.ac.at/envstreams/sensors/observation/instant/{↩
                            TIMESTAMP}";
                    r2rml:termType
                       r2rml:IRI ];
            r2rml:predicate
               ssn:observationResultTime ],
            [ r2rml:objectMap
                    [ r2rml:constant
                        "http://ldlab.ifs.tuwien.ac.at/envstreams/ontology/BikeStation";
                    r2rml:termType
                       r2rml:IRI ];
            r2rml:predicate
               ssn:FeatureOfInterest ];
    r2rml:subjectMap
            [ r2rml:class
               ont:Observation;
            r2rml:template
                "http://ldlab.ifs.tuwien.ac.at/envstreams/sensors/citybike/{id}/↩
                    freebikesobservation/{!TIMESTAMP}" ].
rml:outputfreebikes
    rm:logicalSource
            [ rm:iterator
               "//station";
            rm:referenceFormulation
               ql:XPath;
            rm:source
               "http://dynamisch.citybikewien.at/citybike_xml.php" ];
    r2rml:predicateObjectMap
            [ r2rml:objectMap
                    [ r2rml:template
                        "http://ldlab.ifs.tuwien.ac.at/envstreams/sensors/citybike/{id}/↩
                            freebikesobservation/output/obsvalue/{TIMESTAMP}";
                    r2rml:termType
                       r2rml:IRI ];
            r2rml:predicate
               ssn:hasValue ],
            [ r2rml:objectMap
                    [ r2rml:template
                        "http://ldlab.ifs.tuwien.ac.at/envstreams/sensors/citybike/{id}";
                    r2rml:termType
                       r2rml:IRI ];
            r2rml:predicate
               ssn:isProducedBy ];
    r2rml:subjectMap
            [ r2rml:class
               ont:Output;
            r2rml:template
                "http://ldlab.ifs.tuwien.ac.at/envstreams/sensors/citybike/{id}/↩
                    freebikesobservation/output/{!TIMESTAMP}" ].
rml:obsvaluefreebikes
    rm:logicalSource
            [ rm:iterator
               "//station";
            rm:referenceFormulation
               ql:XPath;
            rm:source
               "http://dynamisch.citybikewien.at/citybike_xml.php" ];
    r2rml:predicateObjectMap
            [ r2rml:objectMap
               [ rm:reference "free_bikes";
                 r2rml:datatype xsd:int ];
            r2rml:predicate
               qudt:numericValue ];
    r2rml:subjectMap
            [ r2rml:class
               ssn:ObservationValue;
            r2rml:template
                "http://ldlab.ifs.tuwien.ac.at/envstreams/sensors/citybike/{id}/↩
                    freebikesobservation/output/obsvalue/{!TIMESTAMP}" ].
rml:availableboxes
    rm:logicalSource
```

```
            [ rm:iterator
               "//station";
            rm:referenceFormulation
               ql:XPath;
            rm:source
               "http://dynamisch.citybikewien.at/citybike_xml.php" ];
    r2rml:predicateObjectMap
            [ r2rml:objectMap
                    [ r2rml:template
                        "http://ldlab.ifs.tuwien.ac.at/envstreams/sensors/citybike/{id}/↩
                             availableboxesobservation/{TIMESTAMP}";
                    r2rml:termType
                       r2rml:IRI ];
            r2rml:predicate
               ssn:madeObservation ];
    r2rml:subjectMap
            [ r2rml:class
               ont:Sensor;
            r2rml:template
               "http://ldlab.ifs.tuwien.ac.at/envstreams/sensors/citybike/{id}" ].
rml:observationavailableboxes
    rm:logicalSource
            [ rm:iterator
               "//station";
            rm:referenceFormulation
               ql:XPath;
            rm:source
               "http://dynamisch.citybikewien.at/citybike_xml.php" ];
    r2rml:predicateObjectMap
            [ r2rml:objectMap
                    [ r2rml:template
                        "http://ldlab.ifs.tuwien.ac.at/envstreams/sensors/citybike/{id}/↩
                             availableboxesobservation/output/{TIMESTAMP}";
                    r2rml:termType
                       r2rml:IRI ];
            r2rml:predicate
               ssn:observationResult ],
            [ r2rml:objectMap
                    [ r2rml:constant
                        "http://ldlab.ifs.tuwien.ac.at/envstreams/ontology/AvailableBoxes";
                    r2rml:termType
                       r2rml:IRI ];
            r2rml:predicate
               ssn:observedProperty ],
            [ r2rml:objectMap
                    [ r2rml:template
                        "http://ldlab.ifs.tuwien.ac.at/envstreams/sensors/observation/instant/{↩
                             TIMESTAMP}";
                    r2rml:termType
                       r2rml:IRI ];
            r2rml:predicate
               ssn:observationResultTime ],
            [ r2rml:objectMap
                    [ r2rml:constant
                        "http://ldlab.ifs.tuwien.ac.at/envstreams/ontology/BikeStation";
                    r2rml:termType
                       r2rml:IRI ];
            r2rml:predicate
               ssn:FeatureOfInterest ],
            [ r2rml:objectMap
                    [ r2rml:constant
                        "http://sweet.jpl.nasa.gov/2.3/reprSciUnits.owl#dimensionlessUnit";
                    r2rml:termType
                       r2rml:IRI ];
            r2rml:predicate
               rel:hasUnit ],
            [ r2rml:objectMap
                    [ r2rml:constant
                       "http://ldlab.ifs.tuwien.ac.at/envstreams/citybikestream";
                    r2rml:termType
                       r2rml:IRI ];
            r2rml:predicate
               cube:DataSet ];
```

```
    r2rml:subjectMap
            [ r2rml:class
               ont:Observation;
            r2rml:template
                "http://ldlab.ifs.tuwien.ac.at/envstreams/sensors/citybike/{id}/↩
                     availableboxesobservation/{!TIMESTAMP}" ].
rml:outputavailableboxes
    rm:logicalSource
            [ rm:iterator
               "//station";
            rm:referenceFormulation
               ql:XPath;
            rm:source
               "http://dynamisch.citybikewien.at/citybike_xml.php" ];
    r2rml:predicateObjectMap
            [ r2rml:objectMap
                    [ r2rml:template
                        "http://ldlab.ifs.tuwien.ac.at/envstreams/sensors/citybike/{id}/↩
                             availableboxesobservation/output/obsvalue/{TIMESTAMP}";
                    r2rml:termType
                       r2rml:IRI ];
            r2rml:predicate
               ssn:hasValue ],
            [ r2rml:objectMap
                    [ r2rml:template
                        "http://ldlab.ifs.tuwien.ac.at/envstreams/sensors/citybike/{id}";
                    r2rml:termType
                       r2rml:IRI ];
            r2rml:predicate
               ssn:isProducedBy ];
    r2rml:subjectMap
            [ r2rml:class
               ont:Output;
            r2rml:template
                "http://ldlab.ifs.tuwien.ac.at/envstreams/sensors/citybike/{id}/↩
                     availableboxesobservation/output/{!TIMESTAMP}" ].
rml:obsvalueavailableboxes
    rm:logicalSource
            [ rm:iterator
               "//station";
            rm:referenceFormulation
               ql:XPath;
            rm:source
               "http://dynamisch.citybikewien.at/citybike_xml.php" ];
    r2rml:predicateObjectMap
            [ r2rml:objectMap
               [ rm:reference "boxes";
                r2rml:datatype xsd:int ];
            r2rml:predicate
               qudt:numericValue ];
    r2rml:subjectMap
            [ r2rml:class
               ssn:ObservationValue;
            r2rml:template
                "http://ldlab.ifs.tuwien.ac.at/envstreams/sensors/citybike/{id}/↩
                     availableboxesobservation/output/obsvalue/{!TIMESTAMP}" ].
rml:freeboxes
    rm:logicalSource
            [ rm:iterator
               "//station";
            rm:referenceFormulation
               ql:XPath;
            rm:source
               "http://dynamisch.citybikewien.at/citybike_xml.php" ];
    r2rml:predicateObjectMap
            [ r2rml:objectMap
                    [ r2rml:template
                        "http://ldlab.ifs.tuwien.ac.at/envstreams/sensors/citybike/{id}/↩
                             freeboxesobservation/{TIMESTAMP}";
                    r2rml:termType
                       r2rml:IRI ];
            r2rml:predicate
               ssn:madeObservation ];
```

```
    r2rml:subjectMap
            [ r2rml:class
               ont:Sensor;
            r2rml:template
               "http://ldlab.ifs.tuwien.ac.at/envstreams/sensors/citybike/{id}" ].
rml:observationfreeboxes
    rm:logicalSource
            [ rm:iterator
               "//station";
            rm:referenceFormulation
               ql:XPath;
            rm:source
               "http://dynamisch.citybikewien.at/citybike_xml.php" ];
    r2rml:predicateObjectMap
            [ r2rml:objectMap
                    [ r2rml:template
                        "http://ldlab.ifs.tuwien.ac.at/envstreams/sensors/citybike/{id}/↩
                             freeboxesobservation/output/{TIMESTAMP}";
                    r2rml:termType
                       r2rml:IRI ];
            r2rml:predicate
               ssn:observationResult ],
            [ r2rml:objectMap
                    [ r2rml:constant
                        "http://ldlab.ifs.tuwien.ac.at/envstreams/ontology/FreeBoxes";
                    r2rml:termType
                       r2rml:IRI ];
            r2rml:predicate
               ssn:observedProperty ],
            [ r2rml:objectMap
                    [ r2rml:template
                        "http://ldlab.ifs.tuwien.ac.at/envstreams/sensors/observation/instant/{↩
                             TIMESTAMP}";
                    r2rml:termType
                       r2rml:IRI ];
            r2rml:predicate
               ssn:observationResultTime ],
            [ r2rml:objectMap
                    [ r2rml:constant
                        "http://ldlab.ifs.tuwien.ac.at/envstreams/ontology/BikeStation";
                    r2rml:termType
                       r2rml:IRI ];
            r2rml:predicate
               ssn:FeatureOfInterest ],
            [ r2rml:objectMap
                    [ r2rml:constant
                        "http://sweet.jpl.nasa.gov/2.3/reprSciUnits.owl#dimensionlessUnit";
                    r2rml:termType
                       r2rml:IRI ];
            r2rml:predicate
               rel:hasUnit ],
            [ r2rml:objectMap
                    [ r2rml:constant
                       "http://ldlab.ifs.tuwien.ac.at/envstreams/citybikestream";
                    r2rml:termType
                       r2rml:IRI ];
            r2rml:predicate
               cube:DataSet ];
    r2rml:subjectMap
            [ r2rml:class
               ont:Observation;
            r2rml:template
                "http://ldlab.ifs.tuwien.ac.at/envstreams/sensors/citybike/{id}/↩
                     freeboxesobservation/{!TIMESTAMP}" ].
rml:outputfreeboxes
    rm:logicalSource
            [ rm:iterator
               "//station";
            rm:referenceFormulation
               ql:XPath;
            rm:source
               "http://dynamisch.citybikewien.at/citybike_xml.php" ];
    r2rml:predicateObjectMap
```

```
            [ r2rml:objectMap
                    [ r2rml:template
                        "http://ldlab.ifs.tuwien.ac.at/envstreams/sensors/citybike/{id}/↩
                             freeboxesobservation/output/obsvalue/{TIMESTAMP}";
                    r2rml:termType
                       r2rml:IRI ];
            r2rml:predicate
               ssn:hasValue ],
            [ r2rml:objectMap
                    [ r2rml:template
                        "http://ldlab.ifs.tuwien.ac.at/envstreams/sensors/citybike/{id}";
                    r2rml:termType
                       r2rml:IRI ];
            r2rml:predicate
               ssn:isProducedBy ];
    r2rml:subjectMap
            [ r2rml:class
               ont:Output;
            r2rml:template
                "http://ldlab.ifs.tuwien.ac.at/envstreams/sensors/citybike/{id}/↩
                     freeboxesobservation/output/{!TIMESTAMP}" ].
rml:time
    rm:logicalSource
            [ rm:iterator
               "//station";
            rm:referenceFormulation
               ql:XPath;
            rm:source
               "http://dynamisch.citybikewien.at/citybike_xml.php" ];
    r2rml:predicateObjectMap
            [ r2rml:objectMap
               [ r2rml:template "{TIMESTAMP}";
                 r2rml:datatype xsd:dateTime ];
            r2rml:predicate
               time:inXSDDateTime ];
    r2rml:subjectMap
            [ r2rml:class
               time:Instant;
            r2rml:template
                "http://ldlab.ifs.tuwien.ac.at/envstreams/sensors/observation/instant/{!TIMESTAMP↩
                     }" ].
rml:obsvaluefreeboxes
    rm:logicalSource
            [ rm:iterator
               "//station";
            rm:referenceFormulation
               ql:XPath;
            rm:source
               "http://dynamisch.citybikewien.at/citybike_xml.php" ];
    r2rml:predicateObjectMap
            [ r2rml:objectMap
               [ rm:reference "free_boxes";
                 r2rml:datatype xsd:int ];
            r2rml:predicate
               qudt:numericValue ];
    r2rml:subjectMap
            [ r2rml:class
               ssn:ObservationValue;
            r2rml:template
                "http://ldlab.ifs.tuwien.ac.at/envstreams/sensors/citybike/{id}/↩
                     freeboxesobservation/output/obsvalue/{!TIMESTAMP}" ].
```

Listing E.2: RML mapping to create stream data for the citybike use case

F Continuous queries for the citybike use case

```
REGISTER STREAM citybikequery AS
CONSTRUCT { ?s rdfs:label ?label .
    ?s geo:lat ?lat .
    ?s geo:long ?lon .
    ?s ssn:madeObservation ?o1 .
        ?o1 ei:FreeBoxes ?fboxes .
        ?o1 time:inXSDDateTime ?time .
    ?s ssn:madeObservation ?o2 .
        ?o2 ei:FreeBikes ?fbikes .
        ?o2 time:inXSDDateTime ?time .
    ?s ssn:madeObservation ?o3 .
        ?o3 ei:AvailableBoxes ?avail .
        ?o3 time:inXSDDateTime ?time .
 }
 FROM <http://linkedwidgets.org/StreamingWidgets/cityBikeStatic.rdf>
 FROM STREAM ↩
   <http://ldlab.ifs.tuwien.ac.at/envstreams/citybikestream> ↩
   [RANGE 2m STEP 2m]
 WHERE { ?s ssn:madeObservation ?o1 .
     ?s rdfs:label ?label .
         ?o1 ssn:observedProperty ei:FreeBoxes .
         ?o1 ssn:observationResult [ ssn:hasValue [ ↩
           qudt:numericValue ?fboxes ] ] .
         ?o1 ssn:observationResultTime [ time:inXSDDateTime ?time ↩
           ] .
     ?s geo:location [ geo:lat ?lat ] .
     ?s geo:location [ geo:long ?lon ] .
     ?s ssn:madeObservation ?o2 .
         ?o2 ssn:observedProperty ei:FreeBikes .
         ?o2 ssn:observationResult [ ssn:hasValue [ ↩
           qudt:numericValue ?fbikes ] ] .
         ?o2 ssn:observationResultTime [ time:inXSDDateTime ?time ↩
           ] .
     ?s ssn:madeObservation ?o3 .
         ?o3 ssn:observedProperty ei:AvailableBoxes .
         ?o3 ssn:observationResult [ ssn:hasValue [ ↩
           qudt:numericValue ?avail ] ] .
         ?o3 ssn:observationResultTime [ time:inXSDDateTime ?time ↩
           ] .
 }
```

Listing F.1: C-SPARQL continuous query to retrieve citybike observations. For the sake of brevity we omit prefix definitions

```
REGISTER STREAM citybikeaggdquery AS PREFIX
CONSTRUCT { ?s rdfs:label ?label .
    ?s geo:lat ?lat .
    ?s geo:long ?lon .
    ?s ssn:madeObservation ?o1n .
        ?o1n ei:FreeBoxes ?fboxes .
        ?o1n time:inXSDDateTime ?time .
    ?s ssn:madeObservation ?o2n .
        ?o2n ei:FreeBikes ?fbikes .
        ?o2n time:inXSDDateTime ?time .
    ?s ssn:madeObservation ?o3n .
        ?o3n ei:AvailableBoxes ?avail .
        ?o3n time:inXSDDateTime ?time .
}
FROM <http://linkedwidgets.org/StreamingWidgets/cityBikeStatic.rdf>
FROM STREAM <http://streamreasoning.org/streams/citybikeagghquery> ↩
  [RANGE 24h STEP 24h]
WHERE {
SELECT ?s ?label (xsd:dateTime(CONCAT(SUBSTR(STR(afn:now()), 1, ↩
  17), '00')) as ?time ) (AVG(?fboxesSingle) as ?fboxes) ↩
  (AVG(?fbikesSingle) as ?fbikes) (AVG(?availSingle) as ?avail) ↩
  ?lat ?lon (IRI(CONCAT(STR(?s), '/freeboxesobservation/', ↩
  SUBSTR(STR(afn:now()), 1, 17), '00')) AS ?o1n) ↩
  (IRI(CONCAT(STR(?s), '/freebikesobservation/', ↩
  SUBSTR(STR(afn:now()), 1, 17), '00')) AS ?o2n) ↩
  (IRI(CONCAT(STR(?s), '/availableboxesobservation/', ↩
  SUBSTR(STR(afn:now()), 1, 17), '00')) AS ?o3n) WHERE { ?s ↩
  rdfs:label ?label .
    ?s geo:lat ?lat .
    ?s geo:long ?lon .
    ?s ssn:madeObservation ?o1 .
        ?o1 ei:FreeBoxes ?fboxesSingle .
        ?o1 time:inXSDDateTime ?time1 .
    ?s ssn:madeObservation ?o2 .
        ?o2 ei:FreeBikes ?fbikesSingle .
        ?o2 time:inXSDDateTime ?time1 .
    ?s ssn:madeObservation ?o3 .
        ?o3 ei:AvailableBoxes ?availSingle .
        ?o3 time:inXSDDateTime ?time1 .
} GROUP BY ?s ?time ?label ?lat ?lon ?o1n ?o2n ?o3n }"
```

Listing F.2: C-SPARQL continuous aggregate query to retrieve citybike observations. For the sake of brevity we omit prefix definitions

G Semantic models of citybike use case widgets

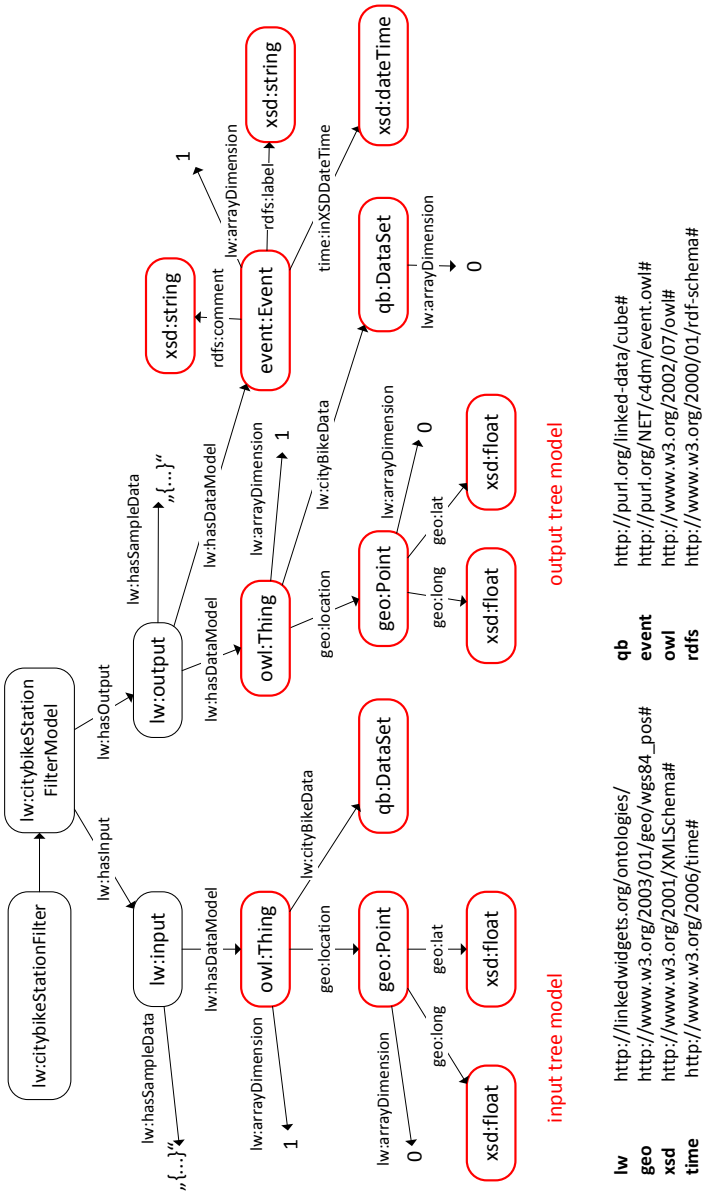
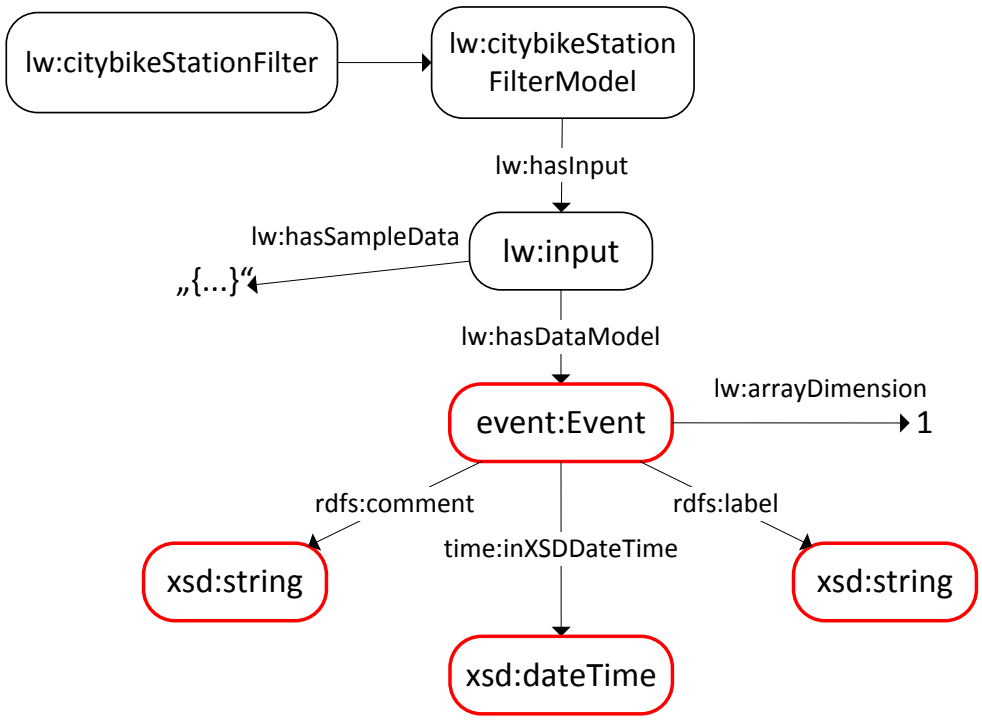


Figure G.1: Semantic model of the Observation Filter widget



lw	http://linkedwidgets.org/ontologies/
xsd	http://www.w3.org/2001/XMLSchema#
event	http://purl.org/NET/c4dm/event.owl#
rdfs	http://www.w3.org/2000/01/rdf-schema#
time	http://www.w3.org/2006/time#

Figure G.2: Semantic model of the Send Event widget

H RML mappings for the route enrichment use case

```
@prefix rml: <http://semweb.mmlab.be/ns/rml#>.
@prefix rr: <http://www.w3.org/ns/r2rml#>.
@prefix rm: <http://pebbie.org/mashup/rml-source/>.
@prefix ql: <http://semweb.mmlab.be/ns/ql#>.
@prefix wgs: <http://www.w3.org/2003/01/geo/wgs84_pos#>.
@prefix xsd: <http://www.w3.org/2001/XMLSchema/>.
@prefix ont: <http://ldlab.ifs.tuwien.ac.at/envstreams/ontology/>.
rm:Mapping1
    rml:logicalSource
            [ rml:iterator
               "$.stations[*]";
            rml:referenceFormulation
               ql:JSONPath;
            rml:source
                "http://luft.umweltbundesamt.at/pub/map_chart/index.pl?runmode=values_json&↩
                  LAT_START=48.16745932392312&LAT_END=48.3982089367818&LNG_START↩
                  =15.840225219726562&LNG_END=16.837234497070312&MEANTYPE=HMW&COMPONENT=PM10_K" ↩
                  ];
rr:predicateObjectMap [
    rr:objectMap [ rml:reference "MetaInfo.Location" ;
                rr:datatype xsd:string ];
   rr:predicate  <rdfs:comment> ],
            [ rr:objectMap
                    [ rr:template
                       "http://ldlab.ifs.tuwien.ac.at/envstreams/point/{stationid}";
                     rr:termType
                       rr:IRI ];
            rr:predicate
               wgs:location ],
    [ rr:predicate <rdfs:label>;
    rr:objectMap [ rml:reference "MetaInfo.Name";
                rr:datatype xsd:string ];
];
    rr:subjectMap
            [ rr:class
               ont:Sensor;
            rr:template
               "http://ldlab.ifs.tuwien.ac.at/envstreams/sensors/airquality/{!stationid}" ].

rm:Mapping2
    rml:logicalSource
            [ rml:iterator
               "$.stations[*]";
            rml:referenceFormulation
               ql:JSONPath;
            rml:source
                "http://luft.umweltbundesamt.at/pub/map_chart/index.pl?runmode=values_json&↩
                  LAT_START=48.16745932392312&LAT_END=48.3982089367818&LNG_START↩
                  =15.840225219726562&LNG_END=16.837234497070312&MEANTYPE=HMW&COMPONENT=PM10_K" ↩
                  ];
    rr:predicateObjectMap
            [ rr:objectMap
               [ rml:reference "gml$Point.gml$coord.X" ;
                 rr:datatype xsd:string ];
            rr:predicate
               wgs:long ],
            [ rr:objectMap
               [ rml:reference "gml$Point.gml$coord.Y" ;
                 rr:datatype xsd:string ];
            rr:predicate
               wgs:lat ];
    rr:subjectMap
            [ rr:class
               wgs:Point;
            rr:template
               "http://ldlab.ifs.tuwien.ac.at/envstreams/point/{!stationid}" ].


rm:Mapping3
    rml:logicalSource
```

```
            [ rml:iterator
               "$.stations[*]";
            rml:referenceFormulation
               ql:JSONPath;
            rml:source
                "http://luft.umweltbundesamt.at/pub/map_chart/index.pl?runmode=values_json&↩
                  LAT_START=48.16745932392312&LAT_END=48.3982089367818&LNG_START↩
                  =15.840225219726562&LNG_END=16.837234497070312&MEANTYPE=HMW&COMPONENT=NO" ];
rr:predicateObjectMap [
    rr:objectMap [ rml:reference "MetaInfo.Location" ;
                 rr:datatype xsd:string ];
   rr:predicate  <rdfs:comment> ],
            [ rr:objectMap
                    [ rr:template
                       "http://ldlab.ifs.tuwien.ac.at/envstreams/point/{stationid}";
                    rr:termType
                       rr:IRI ];
            rr:predicate
               wgs:location ],
    [ rr:predicate <rdfs:label>;
    rr:objectMap [ rml:reference "MetaInfo.Name";
                 rr:datatype xsd:string ];
];
    rr:subjectMap
            [ rr:class
               ont:Sensor;
            rr:template
               "http://ldlab.ifs.tuwien.ac.at/envstreams/sensors/airquality/{!stationid}" ].

rm:Mapping4
    rml:logicalSource
            [ rml:iterator
               "$.stations[*]";
            rml:referenceFormulation
               ql:JSONPath;
            rml:source
                "http://luft.umweltbundesamt.at/pub/map_chart/index.pl?runmode=values_json&↩
                  LAT_START=48.16745932392312&LAT_END=48.3982089367818&LNG_START↩
                  =15.840225219726562&LNG_END=16.837234497070312&MEANTYPE=HMW&COMPONENT=NO" ];
    rr:predicateObjectMap
            [ rr:objectMap
               [ rml:reference "gml$Point.gml$coord.X" ;
                 rr:datatype xsd:string ];
            rr:predicate
               wgs:long ],
            [ rr:objectMap
               [ rml:reference "gml$Point.gml$coord.Y" ;
                 rr:datatype xsd:string ];
            rr:predicate
               wgs:lat ];
    rr:subjectMap
            [ rr:class
               wgs:Point;
            rr:template
               "http://ldlab.ifs.tuwien.ac.at/envstreams/point/{!stationid}" ].



rm:Mapping5
    rml:logicalSource
            [ rml:iterator
               "$.stations[*]";
            rml:referenceFormulation
               ql:JSONPath;
            rml:source
                "http://luft.umweltbundesamt.at/pub/map_chart/index.pl?runmode=values_json&↩
                  LAT_START=48.16745932392312&LAT_END=48.3982089367818&LNG_START↩
                  =15.840225219726562&LNG_END=16.837234497070312&MEANTYPE=HMW&COMPONENT=NO2" ];
rr:predicateObjectMap [
    rr:objectMap [ rml:reference "MetaInfo.Location" ;
                 rr:datatype xsd:string ];
   rr:predicate  <rdfs:comment> ],
            [ rr:objectMap
```

```
                    [ rr:template
                       "http://ldlab.ifs.tuwien.ac.at/envstreams/point/{stationid}";
                      rr:termType
                       rr:IRI ];
             rr:predicate
                wgs:location ],
    [ rr:predicate <rdfs:label>;
    rr:objectMap [ rml:reference "MetaInfo.Name";
                  rr:datatype xsd:string ];
];
    rr:subjectMap
             [ rr:class
                ont:Sensor;
             rr:template
                "http://ldlab.ifs.tuwien.ac.at/envstreams/sensors/airquality/{!stationid}" ].

rm:Mapping6
    rml:logicalSource
             [ rml:iterator
                "$.stations[*]";
             rml:referenceFormulation
                ql:JSONPath;
             rml:source
                 "http://luft.umweltbundesamt.at/pub/map_chart/index.pl?runmode=values_json&↩
                   LAT_START=48.16745932392312&LAT_END=48.3982089367818&LNG_START↩
                   =15.840225219726562&LNG_END=16.837234497070312&MEANTYPE=HMW&COMPONENT=NO2" ];
    rr:predicateObjectMap
             [ rr:objectMap
                [ rml:reference "gml$Point.gml$coord.X" ;
                  rr:datatype xsd:string ];
             rr:predicate
                wgs:long ],
             [ rr:objectMap
                [ rml:reference "gml$Point.gml$coord.Y" ;
                  rr:datatype xsd:string ];
             rr:predicate
                wgs:lat ];
    rr:subjectMap
             [ rr:class
                wgs:Point;
             rr:template
                "http://ldlab.ifs.tuwien.ac.at/envstreams/point/{!stationid}" ].
```

Listing H.1: RML mapping to create static knowledge for the route enrichment use case

```
@prefix xsd: <http://www.w3.org/2001/XMLSchema#>.
@prefix rml: <http://semweb.mmlab.be/ns/rml#>.
@prefix rr: <http://www.w3.org/ns/r2rml#>.
@prefix rm: <http://pebbie.org/mashup/rml-source/>.
@prefix ql: <http://semweb.mmlab.be/ns/ql#>.
@prefix ssn: <http://purl.oclc.org/NET/ssnx/ssn#>.
@prefix ont: <http://ldlab.ifs.tuwien.ac.at/envstreams/ontology/>.
@prefix rel: <http://sweet.jpl.nasa.gov/2.3/relaSci.owl#>.
@prefix cube: <http://purl.org/linked-data/cube#>.
@prefix qudt: <http://qudt.org/schema/qudt#>.
@prefix time: <http://www.w3.org/2006/time#>.

rm:pm10
    rml:logicalSource
            [ rml:iterator
               "$.stations[*]";
            rml:referenceFormulation
               ql:JSONPath;
            rml:source
                "http://luft.umweltbundesamt.at/pub/map_chart/index.pl?runmode=values_json&↩
                  LAT_START=48.16745932392312&LAT_END=48.3982089367818&LNG_START↩
                  =15.840225219726562&LNG_END=16.837234497070312&MEANTYPE=HMW&COMPONENT=PM10_K" ↩
                  ];
    rr:predicateObjectMap
            [ rr:objectMap
                    [ rr:template
                        "http://ldlab.ifs.tuwien.ac.at/envstreams/sensors/airquality/{stationid}/↩
                          pm10observation/{TIMESTAMP}";
                      rr:termType
                         rr:IRI ];
            rr:predicate
               ssn:madeObservation ];
    rr:subjectMap
            [ rr:class
               ont:Sensor;
            rr:template
                "http://ldlab.ifs.tuwien.ac.at/envstreams/sensors/airquality/{!stationid}" ].
rm:observationpm10
    rml:logicalSource
            [ rml:iterator
               "$.stations[*]";
            rml:referenceFormulation
               ql:JSONPath;
            rml:source
                "http://luft.umweltbundesamt.at/pub/map_chart/index.pl?runmode=values_json&↩
                  LAT_START=48.16745932392312&LAT_END=48.3982089367818&LNG_START↩
                  =15.840225219726562&LNG_END=16.837234497070312&MEANTYPE=HMW&COMPONENT=PM10_K" ↩
                  ];
    rr:predicateObjectMap
            [ rr:objectMap
                    [ rr:template
                        "http://ldlab.ifs.tuwien.ac.at/envstreams/sensors/airquality/{stationid}/↩
                          pm10observation/output/{TIMESTAMP}";
                      rr:termType
                         rr:IRI ];
            rr:predicate
               ssn:observationResult ],
            [ rr:objectMap
                    [ rr:constant
                       "http://sweet.jpl.nasa.gov/2.3/propMass.owl#Density";
                      rr:termType
                         rr:IRI ];
            rr:predicate
               ssn:observedProperty ],
            [ rr:objectMap
                    [ rr:template
                        "http://ldlab.ifs.tuwien.ac.at/envstreams/sensors/observation/instant/{↩
                          TIMESTAMP}";
                      rr:termType
                         rr:IRI ];
            rr:predicate
               ssn:observationResultTime ],
```

```
            [ rr:objectMap
                    [ rr:constant
                       "http://sweet.jpl.nasa.gov/2.3/matrAerosol.owl#PM10";
                    rr:termType
                       rr:IRI ];
            rr:predicate
               ssn:FeatureOfInterest ],
            [ rr:objectMap
                    [ rr:constant
                        "http://sweet.jpl.nasa.gov/2.3/reprSciUnits.owl#kilogramPerMeterCubed";
                    rr:termType
                       rr:IRI ];
            rr:predicate
               rel:hasUnit ],
            [ rr:objectMap
                    [ rr:constant
                       "http://ldlab.ifs.tuwien.ac.at/envstreams/airqualitystream";
                    rr:termType
                       rr:IRI ];
            rr:predicate
               cube:DataSet ];
    rr:subjectMap
            [ rr:class
               ont:Observation;
            rr:template
                "http://ldlab.ifs.tuwien.ac.at/envstreams/sensors/airquality/{!stationid}/↩
                  pm10observation/{!TIMESTAMP}" ].
rm:outputpm10
    rml:logicalSource
            [ rml:iterator
               "$.stations[*]";
            rml:referenceFormulation
               ql:JSONPath;
            rml:source
                "http://luft.umweltbundesamt.at/pub/map_chart/index.pl?runmode=values_json&↩
                  LAT_START=48.16745932392312&LAT_END=48.3982089367818&LNG_START↩
                  =15.840225219726562&LNG_END=16.837234497070312&MEANTYPE=HMW&COMPONENT=PM10_K" ↩
                  ];
    rr:predicateObjectMap
            [ rr:objectMap
                    [ rr:template
                        "http://ldlab.ifs.tuwien.ac.at/envstreams/sensors/airquality/{stationid}/↩
                          pm10observation/output/obsvalue/{TIMESTAMP}";
                    rr:termType
                       rr:IRI ];
            rr:predicate
               ssn:hasValue ],
            [ rr:objectMap
                    [ rr:template
                        "http://ldlab.ifs.tuwien.ac.at/envstreams/sensors/airquality/{stationid}"↩
                          ;
                    rr:termType
                       rr:IRI ];
            rr:predicate
               ssn:isProducedBy ];
    rr:subjectMap
            [ rr:class
               ont:Output;
            rr:template
                "http://ldlab.ifs.tuwien.ac.at/envstreams/sensors/airquality/{!stationid}/↩
                  pm10observation/output/{!TIMESTAMP}" ].
rm:obsvaluepm10
    rml:logicalSource
            [ rml:iterator
               "$.stations[*]";
            rml:referenceFormulation
               ql:JSONPath;
            rml:source
                "http://luft.umweltbundesamt.at/pub/map_chart/index.pl?runmode=values_json&↩
                  LAT_START=48.16745932392312&LAT_END=48.3982089367818&LNG_START↩
                  =15.840225219726562&LNG_END=16.837234497070312&MEANTYPE=HMW&COMPONENT=PM10_K" ↩
                  ];
    rr:predicateObjectMap
```

```
            [ rr:objectMap
               [ rml:reference "value"; rr:datatype xsd:int ];
            rr:predicate
               qudt:numericValue ];
    rr:subjectMap
            [ rr:class
               ssn:ObservationValue;
            rr:template
                "http://ldlab.ifs.tuwien.ac.at/envstreams/sensors/airquality/{!stationid}/↩
                  pm10observation/output/obsvalue/{!TIMESTAMP}" ].

rm:no
    rml:logicalSource
            [ rml:iterator
               "$.stations[*]";
            rml:referenceFormulation
               ql:JSONPath;
            rml:source
                "http://luft.umweltbundesamt.at/pub/map_chart/index.pl?runmode=values_json&↩
                  LAT_START=48.16745932392312&LAT_END=48.3982089367818&LNG_START↩
                  =15.840225219726562&LNG_END=16.837234497070312&MEANTYPE=HMW&COMPONENT=NO" ];
    rr:predicateObjectMap
            [ rr:objectMap
                    [ rr:template
                        "http://ldlab.ifs.tuwien.ac.at/envstreams/sensors/airquality/{stationid}/↩
                          noobservation/{TIMESTAMP}";
                    rr:termType
                       rr:IRI ];
            rr:predicate
               ssn:madeObservation ];
    rr:subjectMap
            [ rr:class
               ont:Sensor;
            rr:template
                "http://ldlab.ifs.tuwien.ac.at/envstreams/sensors/airquality/{!stationid}" ].
rm:observationno
    rml:logicalSource
            [ rml:iterator
               "$.stations[*]";
            rml:referenceFormulation
               ql:JSONPath;
            rml:source
                "http://luft.umweltbundesamt.at/pub/map_chart/index.pl?runmode=values_json&↩
                  LAT_START=48.16745932392312&LAT_END=48.3982089367818&LNG_START↩
                  =15.840225219726562&LNG_END=16.837234497070312&MEANTYPE=HMW&COMPONENT=NO" ];
    rr:predicateObjectMap
            [ rr:objectMap
                    [ rr:constant
                        "http://sweet.jpl.nasa.gov/2.3/reprSciUnits.owl#kilogramPerMeterCubed";
                    rr:termType
                       rr:IRI ];
            rr:predicate
               rel:hasUnit ],
            [ rr:objectMap
                    [ rr:constant
                       "http://ldlab.ifs.tuwien.ac.at/envstreams/airqualitystream";
                    rr:termType
                       rr:IRI ];
            rr:predicate
               cube:DataSet ],
            [ rr:objectMap
                    [ rr:template
                        "http://ldlab.ifs.tuwien.ac.at/envstreams/sensors/airquality/{stationid}/↩
                          noobservation/output/{TIMESTAMP}";
                    rr:termType
                       rr:IRI ];
            rr:predicate
               ssn:observationResult ],
            [ rr:objectMap
                    [ rr:constant
                       "http://sweet.jpl.nasa.gov/2.3/propMass.owl#Density";
                    rr:termType
                       rr:IRI ];
```

```
            rr:predicate
               ssn:observedProperty ],
            [ rr:objectMap
                    [ rr:template
                        "http://ldlab.ifs.tuwien.ac.at/envstreams/sensors/observation/instant/{↩
                          TIMESTAMP}";
                    rr:termType
                       rr:IRI ];
            rr:predicate
               ssn:observationResultTime ],
            [ rr:objectMap
                    [ rr:constant
                       "http://sweet.jpl.nasa.gov/2.3/matrCompound.owl#NO";
                    rr:termType
                       rr:IRI ];
            rr:predicate
               ssn:FeatureOfInterest ];
    rr:subjectMap
            [ rr:class
               ont:Observation;
            rr:template
                "http://ldlab.ifs.tuwien.ac.at/envstreams/sensors/airquality/{!stationid}/↩
                  noobservation/{!TIMESTAMP}" ].
rm:outputno
    rml:logicalSource
            [ rml:iterator
               "$.stations[*]";
            rml:referenceFormulation
               ql:JSONPath;
            rml:source
                "http://luft.umweltbundesamt.at/pub/map_chart/index.pl?runmode=values_json&↩
                  LAT_START=48.16745932392312&LAT_END=48.3982089367818&LNG_START↩
                  =15.840225219726562&LNG_END=16.837234497070312&MEANTYPE=HMW&COMPONENT=NO" ];
    rr:predicateObjectMap
            [ rr:objectMap
                    [ rr:template
                        "http://ldlab.ifs.tuwien.ac.at/envstreams/sensors/airquality/{stationid}/↩
                          noobservation/output/obsvalue/{TIMESTAMP}";
                    rr:termType
                       rr:IRI ];
            rr:predicate
               ssn:hasValue ],
            [ rr:objectMap
                    [ rr:template
                        "http://ldlab.ifs.tuwien.ac.at/envstreams/sensors/airquality/{stationid}"↩
                          ;
                    rr:termType
                       rr:IRI ];
            rr:predicate
               ssn:isProducedBy ];
    rr:subjectMap
            [ rr:class
               ont:Output;
            rr:template
                "http://ldlab.ifs.tuwien.ac.at/envstreams/sensors/airquality/{!stationid}/↩
                  noobservation/output/{!TIMESTAMP}" ].
rm:obsvalueno
    rml:logicalSource
            [ rml:iterator
               "$.stations[*]";
            rml:referenceFormulation
               ql:JSONPath;
            rml:source
                "http://luft.umweltbundesamt.at/pub/map_chart/index.pl?runmode=values_json&↩
                  LAT_START=48.16745932392312&LAT_END=48.3982089367818&LNG_START↩
                  =15.840225219726562&LNG_END=16.837234497070312&MEANTYPE=HMW&COMPONENT=NO" ];
    rr:predicateObjectMap
            [ rr:objectMap
               [ rml:reference "value"; rr:datatype xsd:int ];
            rr:predicate
               qudt:numericValue ];
    rr:subjectMap
            [ rr:class
```

```
            ssn:ObservationValue;
        rr:template
            "http://ldlab.ifs.tuwien.ac.at/envstreams/sensors/airquality/{!stationid}/↩
              noobservation/output/obsvalue/{!TIMESTAMP}" ].

rm:no2
    rml:logicalSource
            [ rml:iterator
              "$.stations[*]";
            rml:referenceFormulation
              ql:JSONPath;
            rml:source
              "http://luft.umweltbundesamt.at/pub/map_chart/index.pl?runmode=values_json&↩
                LAT_START=48.16745932392312&LAT_END=48.3982089367818&LNG_START↩
                =15.840225219726562&LNG_END=16.837234497070312&MEANTYPE=HMW&COMPONENT=NO2" ];
    rr:predicateObjectMap
            [ rr:objectMap
                [ rr:template
                    "http://ldlab.ifs.tuwien.ac.at/envstreams/sensors/airquality/{stationid}/↩
                      no2observation/{TIMESTAMP}";
                rr:termType
                    rr:IRI ];
            rr:predicate
              ssn:madeObservation ];
    rr:subjectMap
            [ rr:class
              ont:Sensor;
            rr:template
              "http://ldlab.ifs.tuwien.ac.at/envstreams/sensors/airquality/{!stationid}" ].
rm:observationno2
    rml:logicalSource
            [ rml:iterator
              "$.stations[*]";
            rml:referenceFormulation
              ql:JSONPath;
            rml:source
              "http://luft.umweltbundesamt.at/pub/map_chart/index.pl?runmode=values_json&↩
                LAT_START=48.16745932392312&LAT_END=48.3982089367818&LNG_START↩
                =15.840225219726562&LNG_END=16.837234497070312&MEANTYPE=HMW&COMPONENT=NO2" ];
    rr:predicateObjectMap
            [ rr:objectMap
                [ rr:template
                    "http://ldlab.ifs.tuwien.ac.at/envstreams/sensors/airquality/{stationid}/↩
                      no2observation/output/{TIMESTAMP}";
                rr:termType
                    rr:IRI ];
            rr:predicate
              ssn:observationResult ],
            [ rr:objectMap
                [ rr:constant
                  "http://sweet.jpl.nasa.gov/2.3/propMass.owl#Density";
                rr:termType
                    rr:IRI ];
            rr:predicate
              ssn:observedProperty ],
            [ rr:objectMap
                [ rr:template
                    "http://ldlab.ifs.tuwien.ac.at/envstreams/sensors/observation/instant/{↩
                      TIMESTAMP}";
                rr:termType
                    rr:IRI ];
            rr:predicate
              ssn:observationResultTime ],
            [ rr:objectMap
                [ rr:constant
                  "http://sweet.jpl.nasa.gov/2.3/matrCompound.owl#NO2";
                rr:termType
                    rr:IRI ];
            rr:predicate
              ssn:FeatureOfInterest ],
            [ rr:objectMap
                [ rr:constant
                    "http://sweet.jpl.nasa.gov/2.3/reprSciUnits.owl#kilogramPerMeterCubed";
```

```
                    rr:termType
                       rr:IRI ];
            rr:predicate
               rel:hasUnit ],
            [ rr:objectMap
                    [ rr:constant
                       "http://ldlab.ifs.tuwien.ac.at/envstreams/airqualitystream";
                    rr:termType
                       rr:IRI ];
            rr:predicate
               cube:DataSet ];
    rr:subjectMap
            [ rr:class
               ont:Observation;
            rr:template
                "http://ldlab.ifs.tuwien.ac.at/envstreams/sensors/airquality/{!stationid}/↩
                  no2observation/{!TIMESTAMP}" ].
rm:outputno2
    rml:logicalSource
            [ rml:iterator
               "$.stations[*]";
            rml:referenceFormulation
               ql:JSONPath;
            rml:source
                "http://luft.umweltbundesamt.at/pub/map_chart/index.pl?runmode=values_json&↩
                  LAT_START=48.16745932392312&LAT_END=48.3982089367818&LNG_START↩
                  =15.840225219726562&LNG_END=16.837234497070312&MEANTYPE=HMW&COMPONENT=NO2" ];
    rr:predicateObjectMap
            [ rr:objectMap
                    [ rr:template
                        "http://ldlab.ifs.tuwien.ac.at/envstreams/sensors/airquality/{stationid}/↩
                          no2observation/output/obsvalue/{TIMESTAMP}";
                    rr:termType
                       rr:IRI ];
            rr:predicate
               ssn:hasValue ],
            [ rr:objectMap
                    [ rr:template
                        "http://ldlab.ifs.tuwien.ac.at/envstreams/sensors/airquality/{stationid}"↩
                          ;
                    rr:termType
                       rr:IRI ];
            rr:predicate
               ssn:isProducedBy ];
    rr:subjectMap
            [ rr:class
               ont:Output;
            rr:template
                "http://ldlab.ifs.tuwien.ac.at/envstreams/sensors/airquality/{!stationid}/↩
                  no2observation/output/{!TIMESTAMP}" ].

rm:obsvalueno2
    rml:logicalSource
            [ rml:iterator
               "$.stations[*]";
            rml:referenceFormulation
               ql:JSONPath;
            rml:source
                "http://luft.umweltbundesamt.at/pub/map_chart/index.pl?runmode=values_json&↩
                  LAT_START=48.16745932392312&LAT_END=48.3982089367818&LNG_START↩
                  =15.840225219726562&LNG_END=16.837234497070312&MEANTYPE=HMW&COMPONENT=NO2" ];
    rr:predicateObjectMap
            [ rr:objectMap
               [ rml:reference "value"; rr:datatype xsd:int ];
            rr:predicate
               qudt:numericValue ];
    rr:subjectMap
            [ rr:class
               ssn:ObservationValue;
            rr:template
                "http://ldlab.ifs.tuwien.ac.at/envstreams/sensors/airquality/{!stationid}/↩
                  no2observation/output/obsvalue/{!TIMESTAMP}" ].
```

```
rm:time
    rml:logicalSource
            [ rml:iterator
               "$.stations[*]";
            rml:referenceFormulation
               ql:JSONPath;
            rml:source
                "http://luft.umweltbundesamt.at/pub/map_chart/index.pl?runmode=values_json&↩
                  LAT_START=48.16745932392312&LAT_END=48.3982089367818&LNG_START↩
                  =15.840225219726562&LNG_END=16.837234497070312&MEANTYPE=HMW&COMPONENT=NO" ];
    rr:predicateObjectMap
            [ rr:objectMap
               [ rr:datatype xsd:dateTime; rr:template "{TIMESTAMP}" ];
            rr:predicate
               time:inXSDDateTime ];
    rr:subjectMap
            [ rr:class
               time:Instant;
            rr:template
                "http://ldlab.ifs.tuwien.ac.at/envstreams/sensors/observation/instant/{!TIMESTAMP↩
                  }" ].
```

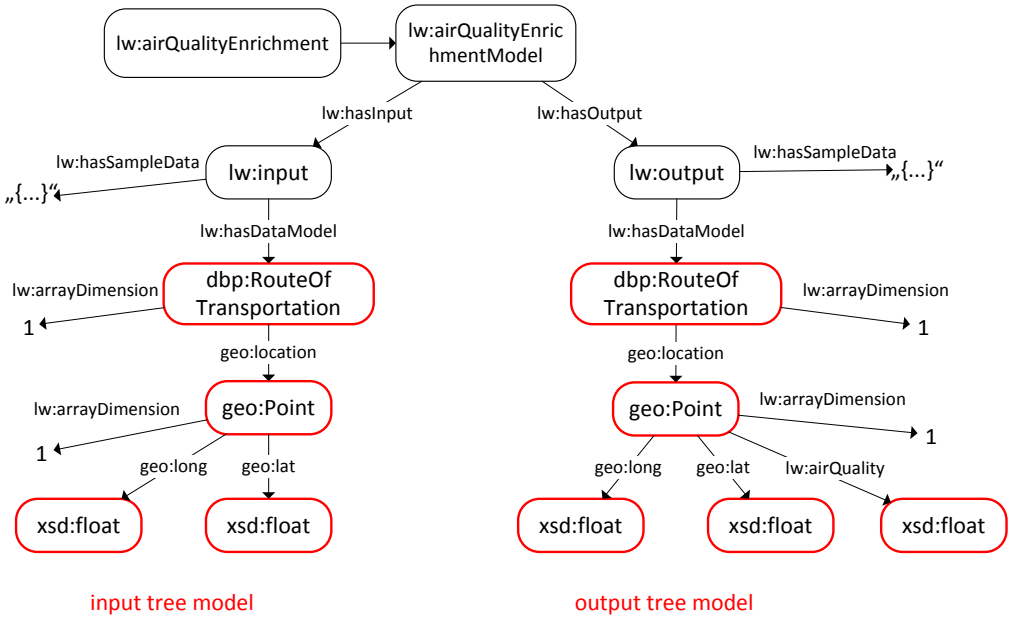
Listing H.2: RML mapping to create stream data for the route enrichment use case

I Continuous queries for the route enrichment use case

```
REGISTER STREAM airqualityquery AS
CONSTRUCT {
    ?s rdfs:label ?label .
    ?s geo:lat ?lat .
    ?s geo:long ?lon .
    ?s ssn:madeObservation ?o1 .
        ?o1 sweetComp:NO2 ?no2 .
        ?o1 time:inXSDDateTime ?time .
    ?s ssn:madeObservation ?o2 .
        ?o2 sweetAir:PM10 ?pm10 .
        ?o2 time:inXSDDateTime ?time .
    ?s ssn:madeObservation ?o3 .
        ?o3 sweetComp:NO ?no .
        ?o3 time:inXSDDateTime ?time .
}
FROM <http://linkedwidgets.org/StreamingWidgets/airQualityStatic.rdf>
FROM STREAM ↩
  <http://ldlab.ifs.tuwien.ac.at/envstreams/airqualitystream> ↩
  [RANGE 5m STEP 5m]
WHERE {
    ?s rdf:type ei:Sensor .
    ?s rdfs:label ?label .
    ?s geo:location [ geo:lat ?lat ] .
    ?s geo:location [ geo:long ?lon ] .
OPTIONAL {
    ?s ssn:madeObservation ?o1 .
        ?o1 ssn:FeatureOfInterest sweetComp:NO2 .
        ?o1 ssn:observationResult [ ssn:hasValue [ ↩
          qudt:numericValue ?no2 ] ] .
        ?o1 ssn:observationResultTime [ time:inXSDDateTime ?time ] .
} OPTIONAL {
    ?s ssn:madeObservation ?o2 .
        ?o2 ssn:FeatureOfInterest sweetAir:PM10 .
        ?o2 ssn:observationResult [ ssn:hasValue [ ↩
          qudt:numericValue ?pm10 ] ].
        ?o2 ssn:observationResultTime [ time:inXSDDateTime ?time ] .
} OPTIONAL {
    ?s ssn:madeObservation ?o3 .
        ?o3 ssn:FeatureOfInterest sweetComp:NO .
        ?o3 ssn:observationResult [ ssn:hasValue [ ↩
          qudt:numericValue ?no ] ] .
        ?o3 ssn:observationResultTime [ time:inXSDDateTime ?time ] .
} }
```

Listing I.1: C-SPARQL continuous query to retrieve air quality observations. For the sake of brevity we omit prefix definitions

J Semantic model of route enrichment use case widget



lw	http://linkedwidgets.org/ontologies/
geo	http://www.w3.org/2003/01/geo/wgs84_pos#
xsd	http://www.w3.org/2001/XMLSchema#
dbp	http://dbpedia.org/ontology/Place

Figure J.1: Semantic model of the Air Quality Enrichment widget

K Construct query cascade

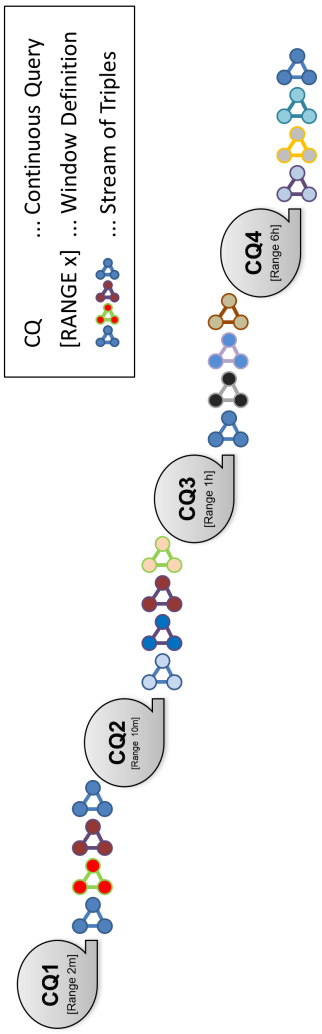


Figure K.1: Construct query cascade used for the evaluation. CQ1–CQ4 are continuous queries with different window definitions. CQ2–CQ4 consist of aggregate functions and consume the output of their preceding queries, that is, a stream of RDF triples.

List of Figures

List of Tables

List of Listings

List of Abbreviations

API Application Programming Interface.

BSBM Berlin SPARQL Benchmark.

C-SPARQL Continuous SPARQL.

CASSRAM Context-aware Sensor Search, Selection, and Ranking Model.

CEP Complex Event Processing.

CPU Central Processing Unit.

CQELS Continuous Query Evaluation over Linked Streams.

CSRBench Correctness checking Benchmark for Streaming RDF/SPARQL.

DBMS Database Management System.

DBPSP DBpedia SPARQL Benchmark.

DSD Data Structure Definition.

DSMS Data Stream Management System.

DUL DOLCE+DnS Ultralite.

EEA European Environment Agency.

EP-SPARQL Event Processing SPARQL.

ESB Environmental Specimen Bank.

ETALIS Event TrAnsaction Logic Inference System.

EU European Union.

GEO Global Environment Outlook.

GPS Global Positioning System.

GPU Graphics Processing Units.

GSN Global Sensor Networks.

GUI Graphical User Interface.

HTML Hypertext Markup Language.

ICT Information and Communication Technologies.

INSTANS Incremental eNgine for STANding Sparql.

IoT Internet of Things.

IT Information Technology.

JSON-LD JavaScript Object Notation for Linked Data.

JTALIS Java Event TrAnsaction Logic Inference System.

KPI Key Performance Indicator.

LarKC The Large Knowledge Collider.

LOD Linked Open Data.

LSBench Linked Stream Benchmark.

LSD Linked Stream Data.

LUBM Lehigh University Benchmark.

OGC Open Geospatial Consortium.

OWL Web Ontology Language.

PROV-O PROV Ontology.

QB RDF Data Cube Vocabulary.

QUDT Quantities, Units, Dimensions and Data Types Ontology.

R2RML RDB to RDF Mapping Language.

RDF Resource Description Framework.

RDFS Resource Description Framework Schema.

REST Representational State Transfer.

RML RDF Mapping language.

RSP RDF Stream Processing.

SAWSDL Semantic Annotations for WSDL.

SECRET Scope, Content, Report, and Tick.

SNEE Sensor NEtwork Engine.

SPARQL Simple Protocol and RDF Query Language.

SQL Structured Query Language.

SRBench Streaming RDF/SPARQL Benchmark.

SSN Semantic Sensor Network.

SSNO Semantic Sensor Network Ontology.

STO Situation Theory Ontology.

SWEET Semantic Web for Earth and Environmental Terminology.

SWRL Semantic Web Rule Language.

UNEP United Nations Environment Programme.

UNO United Nations Organization.

W3C World Wide Web Consortium.

WHO World Health Organization.

WSDL Web Services Description Language.

WSMO Web Service Modeling Ontology.

YABench Yet Another RDF Stream Processing Benchmark.

Bibliography

[AAB+05] Daniel J. Abadi, Yanif Ahmad, Magdalena Balazinska, Ugur Çetintemel, Mitch Cherniack, Jeong-Hyon Hwang, Wolfgang Lindner, Anurag Maskey, Alex Rasin, Esther Ryvkina, and others. The design of the borealis stream processing engine. In *Second Biennial Conference on Innovative Data Systems Research (CIDR 2005)*, volume 5, pages 277–289, 2005.

[ABB+04] Arvind Arasu, Brian Babcock, Shivnath Babu, John Cieslewicz, Mayur Datar, Keith Ito, Rajeev Motwani, Utkarsh Srivastava, and Jennifer Widom. Stream: The stanford data stream management system. Technical report, 2004.

[ABK+07] Sören Auer, Christian Bizer, Georgi Kobilarov, Jens Lehmann, Richard Cyganiak, and Zachary G. Ives. Dbpedia: A nucleus for a web of open data. In Karl Aberer, Key-Sun Choi, Natasha Fridman Noy, Dean Allemang, Kyung-Il Lee, Lyndon J. B. Nixon, Jennifer Golbeck, Peter Mika, Diana Maynard, Riichiro Mizoguchi, Guus Schreiber, and Philippe Cudré-Mauroux, editors, *The Semantic Web, 6th International Semantic Web Conference, 2nd Asian Semantic Web Conference, ISWC 2007 + ASWC 2007, Busan, Korea, November 11-15, 2007.*, volume 4825 of *Lecture Notes in Computer Science*, pages 722–735. Springer, 2007.

[ABM02] Serge Abiteboul, Omar Benjelloun, and Tova Milo. Web services and data integration. In Tok Wang Ling, Umeshwar Dayal, Elisa Bertino, Wee Keong Ng, and Angela Goh, editors, *3rd International Conference on Web Information Systems Engineering, WISE 2002, Singapore, December 12-14, 2002, Proceedings*, pages 3–6. IEEE Computer Society, 2002.

[ABW06] Arvind Arasu, Shivnath Babu, and Jennifer Widom. The CQL continuous query language: semantic foundations and query execution. *The VLDB Journal*, 15(2):121–142, 2006.

[ACÇ+03a] Daniel J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, C. Erwin, E. Galvez, M. Hatoun, J.-h Hwang, A. Maskey, A. Rasin, A. Singer, M. Stonebraker, N. Tatbul, Y. Xing, R. Yan, and S. Zdonik. Aurora: A data stream management system. In *Proceedings of the 2003*

ACM SIGMOD International Conference on Management of Data, page 666, 2003.

[ACÇ+03b] Daniel J. Abadi, Don Carney, Ugur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. Aurora: a new model and architecture for data stream management. *The VLDB Journal*, 12(2):120–139, 2003.

[ACG+04] Arvind Arasu, Mitch Cherniack, Eduardo Galvez, David Maier, Anurag S. Maskey, Esther Ryvkina, Michael Stonebraker, and Richard Tibbetts. Linear road: A stream data management benchmark. In *Proceedings of the Thirtieth International Conference on Very Large Data Bases - Volume 30*, VLDB '04, pages 480–491. VLDB Endowment, 2004.

[ACKM04] Gustavo Alonso, Fabio Casati, Harumi Kuno, and Vijay Machiraju. *Web services*. Springer, Berlin, Heidelberg, 2004.

[AFR+11] Darko Anicic, Paul Fodor, Sebastian Rudolph, Roland Stühmer, Nenad Stojanovic, and Rudi Studer. ETALIS: Rule-Based reasoning in event processing. In Sven Helmer, Alexandra Poulovassilis, and Fatos Xhafa, editors, *Reasoning in Event-Based Distributed Systems*, volume 347 of *Studies in Computational Intelligence*, pages 99–124. Springer, Berlin, Heidelberg, 2011.

[AFRS11] Darko Anicic, Paul Fodor, Sebastian Rudolph, and Nenad Stojanovic. EP-SPARQL: a unified language for event processing and stream reasoning. In *Proceedings of the 20th International Conference on World Wide Web*, WWW '11, page 635–644, New York, NY, USA, 2011. ACM.

[AHS06] Karl Aberer, Manfred Hauswirth, and Ali Salehi. A middleware for fast and flexible sensor network deployment. In *Proceedings of the 32nd international conference on Very large data bases*, pages 1199–1202, 2006.

[AKK+03] Marcelo Arenas, Vasiliki Kantere, Anastasios Kementsietsidis, Iluju Kiringa, Renée J Miller, and John Mylopoulos. The hyperion project: from data integration to data coordination. *ACM SIGMOD Record*, 32(3):53–58, 2003.

[BBC+10a] Davide Francesco Barbieri, Daniele Braga, Stefano Ceri, Emanuele Della Valle, and Michael Grossniklaus. Incremental reasoning on streams and rich background knowledge. In *Proceedings of the 7th International Conference on The Semantic Web: Research and Applications - Volume Part I*, ESWC'10, pages 1–15, Berlin, Heidelberg, 2010. Springer-Verlag.

[BBC+10b] Davide Francesco Barbieri, Daniele Braga, Stefano Ceri, Emanuele Della Valle, and Michael Grossniklaus. C-SPARQL: a continuous query language

for RDF data streams. *International Journal of Semantic Computing*, 4(01):3–25, 2010.

[BBC+10c] Davide Francesco Barbieri, Daniele Braga, Stefano Ceri, Emanuele Della Valle, and Michael Grossniklaus. Querying RDF streams with C-SPARQL. *SIGMOD Rec.*, 39(1):20–26, September 2010.

[BBCG10] Davide Francesco Barbieri, Daniele Braga, Stefano Ceri, and Michael Grossniklaus. An execution environment for C-SPARQL queries. In *Proceedings of the 13th International Conference on Extending Database Technology*, pages 441–452. ACM, 2010.

[BBD+02] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Jennifer Widom. Models and issues in data stream systems. In *Proceedings of the Twenty-first ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '02, pages 1–16, New York, NY, USA, 2002. ACM.

[BCD+14] Marco Balduini, Irene Celino, Daniele Dell'Aglio, Emanuele Della Valle, Yi Huang, Tony Lee, Seon-Ho Kim, and Volker Tresp. Reality mining on micropost streams – deductive and inductive reasoning for personalized and location-based recommendations. *Semantic Web*, 5(5):341–356, 2014.

[BCVB01] Sonia Bergamaschi, Silvana Castano, Maurizio Vincini, and Domenico Beneventano. Semantic integration of heterogeneous information sources. *Data Knowl. Eng.*, 36(3):215–249, 2001.

[BDD+10] Irina Botan, Roozbeh Derakhshan, Nihal Dindar, Laura Haas, Renée J. Miller, and Nesime Tatbul. SECRET: A model for analysis of the execution semantics of stream processing systems. *Proc. VLDB Endow.*, 3(1-2):232–243, September 2010.

[BDS08] Djamal Benslimane, Schahram Dustdar, and Amit P. Sheth. Services mashups: The new generation of web applications. *IEEE Internet Computing*, 12(5):13–15, 2008.

[BDVD+13] Marco Balduini, Emanuele Della Valle, Daniele Dell'Aglio, Mikalai Tsytsarau, Themis Palpanas, and Cristian Confalonieri. Social listening of city scale events using the streaming linked data framework. In *The Semantic Web–ISWC 2013*, page 1–16. Springer, Berlin, Heidelberg, 2013.

[Ber06] Tim Berners-Lee. Linked data - design issues. Technical report, W3C, 2006.

[BETL12] Katrin Braunschweig, Julian Eberius, Maik Thiele, and Wolfgang Lehner. The state of open data - limits of current open data platforms. In *Proceedings of the 21st World Wide Web Conference 2012, Web Science Track at WWW'12, Lyon, France, April 16-20, 2012*. ACM, 2012.

[BJBB+97] Roberto J Bayardo Jr, William Bohrer, Richard Brice, Andrzej Cichocki, Jerry Fowler, Abdelsalam Helal, Vipul Kashyap, Tomasz Ksiezyk, Gale Martin, Marian Nodine, et al. Infosleuth: agent-based semantic integration of information in open and dynamic environments. In *ACM SIGMOD Record*, volume 26, pages 195–206. ACM, 1997.

[BL03] Tim Berners-Lee. WWW past & future. `http://www.w3.org/2003/Talks/0922-rsoc-tbl/`, 2003. accessed 16 June 2016.

[BP10] Payam M. Barnaghi and Mirko Presser. Publishing linked sensor data. In Taylor et al. [TAR10].

[BS09] Christian Bizer and Andreas Schultz. The berlin SPARQL benchmark. *International Journal On Semantic Web and Information Systems*, 5(2):1–24, 2009.

[BTW+06] Yijian Bai, Hetal Thakkar, Haixun Wang, Chang Luo, and Carlo Zaniolo. A data stream language and system designed for power and extensibility. In *Proceedings of the 15th ACM international conference on Information and knowledge management*, pages 337–346. ACM, 2006.

[BV13] Marco Balduini and Emanuele Della Valle. A restful interface for RDF stream processors. In Eva Blomqvist and Tudor Groza, editors, *Proceedings of the ISWC 2013 Posters & Demonstrations Track, Sydney, Australia, October 23, 2013*, volume 1035 of *CEUR Workshop Proceedings*, pages 209–212. CEUR-WS.org, 2013.

[CA15] Jean-Paul Calbimonte and Karl Aberer. *The Semantic Web: ESWC 2015 Satellite Events: ESWC 2015 Satellite Events, Portorož, Slovenia, May 31 – June 4, 2015, Revised Selected Papers*, chapter Reactive Processing of RDF Streams of Events, pages 457–468. Springer International Publishing, Cham, 2015.

[CBB+12] Michael Compton, Payam Barnaghi, Luis Bermudez, Raúl García-Castro, Óscar Corcho, Simon Cox, John Graybeal, Manfred Hauswirth, Cory Henson, Arthur Herzog, Vincent Huang, Krzysztof Janowicz, W. David Kelsey, Danh Le-Phuoc, Laurent Lefort, Myriam Leggieri, Holger Neuhaus, Andriy Nikolov, Kevin Page, Alexandre Passant, Amit Sheth, and Kerry Taylor. The SSN ontology of the W3C semantic sensor network incubator group. *Web Semantics: Science, Services and Agents on the World Wide Web*, 17:25–32, December 2012.

[CCD+03] Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, Michael J. Franklin, Joseph M. Hellerstein, Wei Hong, Sailesh Krishnamurthy, Samuel R. Madden, Fred Reiss, and Mehul A. Shah. TelegraphCQ: continuous dataflow processing. In *Proceedings of the 2003 ACM SIGMOD*

international conference on Management of data, pages 668–668. ACM, 2003.

[CCG10] Jean-Paul Calbimonte, Óscar Corcho, and Alasdair J. G. Gray. Enabling ontology-based access to streaming data sources. In *Proceedings of the 9th International Semantic Web Conference on The Semantic Web - Volume Part I*, ISWC'10, page 96–111, Berlin, Heidelberg, 2010. Springer.

[CCJA12] Óscar Corcho, Jean-Paul Calbimonte, Hoyoung Jeung, and Karl Aberer. Enabling query technologies for the semantic sensor web. *Int. J. Semant. Web Inf. Syst.*, 8(1):43–63, January 2012.

[CDBN11] Andrea Caragliu, Chiara Del Bo, and Peter Nijkamp. Smart cities in europe. *Journal of urban technology*, 18(2):65–82, 2011.

[CDTW00] Jianjun Chen, David J. DeWitt, Feng Tian, and Yuan Wang. NiagaraCQ: A scalable continuous query system for internet databases. In *ACM SIGMOD Record*, volume 29, pages 379–390. ACM, 2000.

[CG10] Óscar Corcho and Raúl García-Castro. Five challenges for the semantic sensor web. *Semantic Web*, 1(1-2):121–125, 2010.

[CGJ+02] Chuck Cranor, Yuan Gao, Theodore Johnson, Vlaidslav Shkapenyuk, and Oliver Spatscheck. Gigascope: High performance network monitoring with an SQL interface. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 623–623. ACM, 2002.

[CHS+95] Michael J Carey, Laura M Haas, Peter M Schwarz, Manish Arya, WE Cody, Ronald Fagin, Myron Flickner, Allen W Luniewski, Wayne Niblack, Dragutin Petkovic, et al. Towards heterogeneous multimedia information systems: The garlic approach. In *Research Issues in Data Engineering, 1995: Distributed Object Management, Proceedings. RIDE-DOM'95. Fifth International Workshop on*, pages 124–131. IEEE, 1995.

[CJSS03] Chuck Cranor, Theodore Johnson, Oliver Spataschek, and Vladislav Shkapenyuk. Gigascope: a stream database for network applications. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 647–651. ACM, 2003.

[CM12] Gianpaolo Cugola and Alessandro Margara. Processing flows of information: From data stream to complex event processing. *ACM Comput. Surv.*, 44(3):15, 2012.

[Coh14] Boyd Cohen. The 10 smartest cities in europe. `http://www.fastcoexist.com/3024721/the-10-smartest-cities-in-europe`, 2014.

[CR14] Richard Cyganiak and Dave Reynolds. The RDF data cube vocabulary. W3C recommendation, W3C, January 2014. http://www.w3.org/TR/2014/REC-vocab-data-cube-20140116/.

[CSGL15] Jules Chevalier, Julien Subercaze, Christophe Gravier, and Frédérique Laforest. Slider: An efficient incremental reasoner. In Timos K. Sellis, Susan B. Davidson, and Zachary G. Ives, editors, *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, pages 1081–1086. ACM, 2015.

[CX05] Isabel F. Cruz and Huiyong Xiao. The role of ontologies in data integration. *Engineering Intelligent Systems*, 13(4):245–252, 12 2005.

[DBDV13] Daniele Dell'Aglio, Marco Balduini, and Emanuele Della Valle. On the need to include functional testing in RDF stream engine benchmarks. In *1st International Workshop On Benchmarking RDF Systems (BeRSys 2013)*, page 10. 2013.

[DCB+13] Daniele Dell'Aglio, Jean-Paul Calbimonte, Marco Balduini, Óscar Corcho, and Emanuele Della Valle. On correctness in RDF stream processor benchmarking. In *The Semantic Web - ISWC 2013 - 12th ISWC, Sydney, NSW, Australia, October 21-25, 2013, Proceedings, Part II*, pages 326–342. Springer, 2013.

[DCDV13] Daniele Dell'Aglio, Irene Celino, and Emanuele Della Valle. *Semantic Mashups: Intelligent Reuse of Web Resources*, chapter Urban Mashups, pages 287–319. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[DCS12] Souripriya Das, Richard Cyganiak, and Seema Sundara. R2RML: RDB to RDF mapping language. W3C recommendation, W3C, September 2012. http://www.w3.org/TR/2012/REC-r2rml-20120927/.

[DCVC15] Daniele Dell'Aglio, Jean-Paul Calbimonte, Emanuele Della Valle, and Óscar Corcho. Towards a unified language for RDF stream query processing. In Fabien Gandon, Christophe Guéret, Serena Villata, John G. Breslin, Catherine Faron-Zucker, and Antoine Zimmermann, editors, *The Semantic Web: ESWC 2015 Satellite Events - ESWC 2015 Satellite Events Portorož, Slovenia, May 31 - June 4, 2015, Revised Selected Papers*, volume 9341 of *Lecture Notes in Computer Science*, pages 353–363. Springer, 2015.

[DDG+15] Soheila Dehghanzadeh, Daniele Dell'Aglio, Shen Gao, Emanuele Della Valle, Alessandra Mileo, and Abraham Bernstein. Approximate continuous query answering over streams and dynamic linked data sets. In Philipp Cimiano, Flavius Frasincar, Geert-Jan Houben, and Daniel Schwabe, editors, *Engineering the Web in the Big Data Era - 15th International Conference,*

ICWE 2015, Rotterdam, The Netherlands, June 23-26, 2015, Proceedings, volume 9114 of *Lecture Notes in Computer Science*, pages 307–325. Springer, 2015.

[DMD+15] Soheila Dehghanzadeh, Alessandra Mileo, Daniele Dell'Aglio, Emanuele Della Valle, Shen Gao, and Abraham Bernstein. Online view maintenance for continuous query evaluation. In Aldo Gangemi, Stefano Leonardi, and Alessandro Panconesi, editors, *Proceedings of the 24th International Conference on World Wide Web Companion, WWW 2015, Florence, Italy, May 18-22, 2015 - Companion Volume*, pages 25–26. ACM, 2015.

[DPS13] Rustem Dautov, Iraklis Paraskakis, and Mike Stannett. Utilising stream reasoning techniques to create a self-adaptation framework for cloud environments. In *IEEE/ACM 6th International Conference on Utility and Cloud Computing, UCC 2013*, pages 375–380. IEEE, 2013.

[DSC+14] Anastasia Dimou, Miel Vander Sande, Pieter Colpaert, Ruben Verborgh, Erik Mannens, and Rik Van de Walle. RML: A generic language for integrated RDF mappings of heterogeneous data. In Christian Bizer, Tom Heath, Sören Auer, and Tim Berners-Lee, editors, *Proceedings of the Workshop on Linked Data on the Web co-located with the 23rd International World Wide Web Conference (WWW 2014), Seoul, Korea, April 8, 2014.*, volume 1184 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2014.

[DSS+14] Anastasia Dimou, Miel Vander Sande, Jason Slepicka, Pedro A. Szekely, Erik Mannens, Craig A. Knoblock, and Rik Van de Walle. Mapping hierarchical sources into RDF using the RML mapping language. In *2014 IEEE International Conference on Semantic Computing, Newport Beach, CA, USA, June 16-18, 2014*, pages 151–158. IEEE Computer Society, 2014.

[DVCD+11] Emanuele Della Valle, Irene Celino, Daniele Dell'Aglio, Ralf Grothmann, Florian Steinke, and Volker Tresp. Semantic traffic-aware routing using the larkc platform. *IEEE Internet Computing*, 15(6):15–23, Nov 2011.

[EEA15] EEA. The european environment - state and outlook 2015. `http://www.eea.europa.eu/soer-2015/synthesis/report/action-download-pdf`, 2015.

[EFGK03] Patrick Th Eugster, Pascal A Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM Computing Surveys (CSUR)*, 35(2):114–131, 2003.

[Eng15] Nicole C Engard. *More library mashups: exploring new ways to deliver library data*. Facet Publishing, 2015.

[Env] Environment Agency. Bathing water quality API. `http://environment.data.gov.uk/bwq/`. accessed 16 June 2016.

[FLGPJ97] Mariano Fernández-López, Asunción Gómez-Pérez, and Natalia Juristo. Methontology: from ontological art towards ontological engineering. In *Proc. Symposium on Ontological Engineering of AAAI*, 1997.

[FLSFGP12] Mariano Fernández-López, Mari Carmen Suárez-Figueroa, and Asunción Gómez-Pérez. *Ontology Engineering in a Networked World*, chapter Ontology Development by Reuse, pages 147–170. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[FMMVA13] Susel Fernandez, Ivan Marsa-Maestre, Juan R. Velasco, and Bernardo Alarcos. Ontology alignment architecture for semantic sensor web integration. *Sensors*, 13(9):12581, 2013.

[Gag07] Michel Gagnon. Ontology-based integration of data sources. In *10th International Conference on Information Fusion, FUSION 2007, Québec, Canada, July 9-12, 2007*, pages 1–8. IEEE, 2007.

[GAM14] Feng Gao, Muhammad Intizar Ali, and Alessandra Mileo. Semantic discovery and integration of urban data streams. In *Proc. of the Fifth Workshop on Semantics for Smarter Cities a Workshop at the 13th International Semantic Web Conference (ISWC 2014)*, pages 15–30, 2014.

[GBG+11] Ixent Galpin, Christian YA Brenninkmeijer, Alasdair JG Gray, Farhana Jabeen, Alvaro AA Fernandes, and Norman W Paton. SNEE: a query processor for wireless sensor networks. *Distributed and Parallel Databases*, 29(1-2):31–85, 2011.

[GFG+08] Nancy B. Grimm, Stanley H. Faeth, Nancy E. Golubiewski, Charles L. Redman, Jianguo Wu, Xuemei Bai, and John M. Briggs. Global change and the ecology of cities. *Science*, 319(5864):756–760, 2008.

[GFM15] José M. Giménez-García, Javier D. Fernández, and Miguel A. Martínez-Prieto. HDT-MR: A scalable solution for RDF compression with HDT and MapReduce. In Fabien Gandon, Marta Sabou, Harald Sack, Claudia d'Amato, Philippe Cudré-Mauroux, and Antoine Zimmermann, editors, *The Semantic Web. Latest Advances and New Domains - 12th European Semantic Web Conference, ESWC 2015, Portoroz, Slovenia, May 31 - June 4, 2015. Proceedings*, volume 9088 of *Lecture Notes in Computer Science*, pages 253–268. Springer, 2015.

[GGK+11] Alasdair J. G. Gray, Raul Garcia-Castro, Kostis Kyzirakos, Manos Karpathiotakis, Jean-Paul Calbimonte, Kevin R. Page, Jason Sadler, Alex Frazer, Ixent Galpin, Alvaro A. A. Fernandes, Norman W. Paton, Óscar Corcho, Manolis Koubarakis, David De Roure, Kirk Martinez, and

Asunción Gómez-Pérez. A semantically enabled service architecture for mashups over streaming and stored data. In Grigoris Antoniou, Marko Grobelnik, Elena Paslaru Bontas Simperl, Bijan Parsia, Dimitris Plexousakis, Pieter De Leenheer, and Jeff Z. Pan, editors, *The Semanic Web: Research and Applications - 8th Extended Semantic Web Conference, ESWC 2011, Heraklion, Crete, Greece, May 29 - June 2, 2011, Proceedings, Part II*, volume 6644 of *Lecture Notes in Computer Science*, pages 300–314. Springer, 2011.

[GÖ03] Lukasz Golab and M Tamer Özsu. Issues in data stream management. *ACM Sigmod Record*, 32(2):5–14, 2003.

[Goe11] Thomas Goetz. Harnessing the power of feedback loops | magazine. `http://www.wired.com/2011/06/ff_feedbackloop/`, 2011.

[GPH05] Yuanbo Guo, Zhengxiang Pan, and Jeff Heflin. LUBM: A benchmark for OWL knowledge base systems. *Web Semant.*, 3(2-3):158–182, October 2005.

[Gru93] Thomas R. Gruber. A translation approach to portable ontology specifications. *Knowledge Acquisition*, 5(2):199–220, June 1993.

[GSK+11] Alasdair J. G. Gray, Jason Sadler, Oles Kit, Kostis Kyzirakos, Manos Karpathiotakis, Jean-Paul Calbimonte, Kevin Page, Raúl García-Castro, Alex Frazer, Ixent Galpin, Alvaro A. A. Fernandes, Norman W. Paton, Óscar Corcho, Manolis Koubarakis, David De Roure, Kirk Martinez, and Asunción Gómez-Pérez. A Semantic Sensor Web for Environmental Decision Support Applications. *Sensors*, 11(12):8855–8887, 2011.

[Har98] Ronnie Harding. *Environmental decision-making: The roles of scientists, engineers, and the public*. Federation Press, 1998.

[HB91] Ali R Hurson and MW Bright. Multidatabase systems: An advanced concept in handling distributed data. *Advances in computers*, 32:149–200, 1991.

[HC03] GH Huang and NB Chang. The perspectives of environmental informatics and systems analysis. *Journal of Environmental Informatics*, 1(1):1–7, 2003.

[HG01] Farshad Hakimpour and Andreas Geppert. Resolving semantic heterogeneity in schema integration. In *FOIS*, pages 297–308, 2001.

[HK] Jesper Hoeksema and Spyros Kotoulas. High-performance distributed stream reasoning using s4. In *First International Workshop on Ordering and Reasoning (OrdRing2011)*.

[HM06] Jane K. Hart and Kirk Martinez. Environmental sensor networks: A revolution in the earth system science? *Earth-Science Reviews*, 78(3–4):177 – 191, 2006.

[HPRR95] LM Hilty, Bernd Page, FJ Radermacher, and W-F Riekert. Environmental informatics as a new discipline of applied computer science. In *Environmental Informatics*, pages 1–11. Springer, 1995.

[HRO06] Alon Halevy, Anand Rajaraman, and Joann Ordille. Data integration: the teenage years. In *Proceedings of the 32nd international conference on Very large data bases*, page 9–16. VLDB Endowment, 2006.

[HS13] Steven Harris and Andy Seaborne. SPARQL 1.1 query language. W3C recommendation, W3C, March 2013. `http://www.w3.org/TR/2013/REC-sparql11-query-20130321/`.

[Hul97] Richard Hull. Managing semantic heterogeneity in databases: A theoretical prospective. In *Proceedings of the Sixteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, PODS '97, page 51–61, New York, NY, USA, 1997. ACM.

[HVMn+11] José M. Hernández-Muñoz, Jesús Bernat Vercher, Luis Muñoz, José A. Galache, Mirko Presser, Luis A. Hernández Gómez, and Jan Pettersson. Smart cities at the forefront of the future internet. In *The future internet*, page 447–462. Springer, Berlin, Heidelberg, 2011.

[ibm] IBM corporation: IBM Infosphere Streams. `http://www-03.ibm.com/software/products/en/infosphere-streams`. accessed 16 June 2016.

[JBS+13] Krzysztof Janowicz, Arne Bröring, Christoph Stasch, Sven Schade, Thomas Everding, and Alejandro Llaves. A restful proxy and data model for linked sensor data. *Int. J. Digital Earth*, 6(3):233–254, 2013.

[JC10] Krzysztof Janowicz and Michael Compton. The stimulus-sensor-observation ontology design pattern and its integration into the semantic sensor network ontology. In Taylor et al. [TAR10].

[KB15a] Robin Keskisärkkä and Eva Blomqvist. *The Semantic Web: ESWC 2015 Satellite Events: ESWC 2015 Satellite Events, Portorož, Slovenia, May 31 – June 4, 2015, Revised Selected Papers*, chapter Supporting Real-Time Monitoring in Criminal Investigations, pages 82–86. Springer International Publishing, Cham, 2015.

[KB15b] Robin Keskisärkkä and Eva Blomqvist. Sharing and reusing continuous queries - expression of interest. In *RDF Stream Processing Workshop, May 31, Portoroz, Slovenia*, 2015.

[KCF12] Srdjan Komazec, Davide Cerri, and Dieter Fensel. Sparkwave: continuous schema-enhanced pattern matching over RDF data streams. In *Proceedings of the 6th ACM International Conference on Distributed Event-Based Systems*, pages 58–68. ACM, 2012.

[Kee13] C. Maria Keet. *Open World Assumption*, pages 1567–1567. Springer New York, New York, NY, 2013.

[Ken11] John Kennedy. Data is the new oil. `http://www.siliconrepublic.com/comms/item/22326-data-is-the-new-oil`, 2011.

[KFFG11] Klemen Kenda, Carolina Fortuna, Blaž Fortuna, and Marko Grobelnik. Videk: a mash-up for environmental intelligence. *AI Mashup Challange, ESWC*, 2011.

[KFM+13] Klemen Kenda, Carolina Fortuna, Alexandra Moraru, Dunja Mladenić, Blaž Fortuna, and Marko Grobelnik. *Semantic Mashups: Intelligent Reuse of Web Resources*, chapter Mashups for the Web of Things, pages 145–169. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[Kuh09] Werner Kuhn. A functional ontology of observation and measurement. In Krzysztof Janowicz, Martin Raubal, and Sergei Levashkin, editors, *GeoSpatial Semantics, Third International Conference, GeoS 2009, Mexico City, Mexico, December 3-4, 2009. Proceedings*, volume 5892 of *Lecture Notes in Computer Science*, pages 26–43. Springer, 2009.

[KVBF07] J. Kopecky, T. Vitvar, C. Bournez, and J. Farrell. SAWSDL: Semantic annotations for WSDL and XML schema. *Internet Computing, IEEE*, 11(6):60–67, 2007.

[KW15] Maxim Kolchin and Peter Wetz. Demo: YABench - yet another RDF stream processing benchmark. In *RDF Stream Processing Workshop co-located with the 11th Extended Semantic Web Conference (ESWC 2014), Portoroz, Slovenia*, 2015.

[KWA+16] Maxim Kolchin, Peter Wetz, Amin Anjomshoaa, Elmar Kiesling, and A Min Tjoa. YABench: A comprehensive framework for RDF stream processor correctness and performance assessment. In *Web Engineering - 16th International Conference on Web Engineering, ICWE 2016, Lugano, Switzerland, June 6-9, 2016, Proceedings*, 2016.

[LBH+] Laurent Lefort, Josh Bobruk, Armin Haller, Kerry Taylor, and Andrew Woolf. A linked sensor data cube for a 100 year homogenised daily temperature dataset. In *In 5th International Workshop on Semantic Sensor Networks (SSN-2012), CEUR-Proceedings*, page 2012.

[LDP+12] Danh Le-Phuoc, Minh Dao-Tran, Minh-Duc Pham, Peter Boncz, Thomas Eiter, and Michael Fink. Linked stream data processing engines: facts and figures. In *The Semantic Web - ISWC 2012 - 11th ISWC, Boston, MA, USA, November 11-15, 2012, Proceedings, Part II*, pages 300–312. Springer, 2012.

[LDPH11] Danh Le-Phuoc, Minh Dao-Tran, Josiane Xavier Parreira, and Manfred Hauswirth. A native and adaptive approach for unified processing of linked streams and linked data. In *Proceedings of the 10th International Conference on The Semantic Web - Volume Part I*, ISWC'11, page 370–388, Berlin, Heidelberg, 2011. Springer.

[Len02] Maurizio Lenzerini. Data integration: A theoretical perspective. In *Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 233–246. ACM, 2002.

[LH14] Sudesh Lutchman and Patrick Hosein. Design and specifications of a repository for real-time open data. In *Proceedings of the 2014 ITU Kaleidoscope Academic Conference: Living in a converged world - Impossible without standards?, Saint-Petersburg, Russia, June 3-5, 2014*, pages 105–110. IEEE, 2014.

[LHPB09] Giusy Di Lorenzo, Hakim Hacid, Hye-Young Paik, and Boualem Benatallah. Data integration in mashups. *SIGMOD Record*, 38(1):59–66, 2009.

[LHT11] Laurent Lefort, Cory Henson, and Kerry Taylor. Semantic sensor network xg final report. *W3C Incubator Group Report 28 June 2011*, 2011.

[LHTW13] Lauren Lefort, Armin Haller, Kerry Taylor, and Andrew Woolf. The ACORN-SAT linked climate dataset. *Semantic Web Journal*, 2013.

[LK14] Alejandro Llaves and Werner Kuhn. An event abstraction layer for the integration of geosensor data. *International Journal of Geographical Information Science*, 28(5):1085–1106, 2014.

[LP09] Danh Le-Phuoc. Sensormasher - publishing and building mashup of sensor data. In Adrian Paschke, Hans Weigand, Wernher Behrendt, Klaus Tochtermann, and Tassilo Pellegrini, editors, *5th International Conference on Semantic Systems, Graz, Austria, September 2-4, 2009. Proceedings.* Verlag der Technischen Universität Graz, 2009.

[LPH09] Danh Le-Phuoc and Manfred Hauswirth. Linked open data in sensor data mashups,. In Kerry Taylor and David De Roure, editors, *Proceedings of the 2nd International Workshop on Semantic Sensor Networks (SSN09), collocated with the 8th International Semantic Web Conference (ISWC-2009), Washington DC, USA, October 26, 2009.*, volume 522 of *CEUR Workshop Proceedings*, pages 1–16. CEUR-WS.org, 2009.

[LPPH+10] Danh Le-Phuoc, Josiane Xavier Parreira, Michael Hausenblas, Yuanbo Han, and Manfred Hauswirth. Live linked open sensor database. In Adrian Paschke, Nicola Henze, and Tassilo Pellegrini, editors, *Proceedings the 6th International Conference on Semantic Systems, I-SEMANTICS 2010, Graz, Austria, September 1-3, 2010*, ACM International Conference Proceeding Series. ACM, 2010.

[LPQLVH13] Danh Le-Phuoc, Hoan Nguyen Mau Quoc, Chan Le-Van, and Manfred Hauswirth. Elastic and scalable processing of linked stream data in the cloud. In Harith Alani, Lalana Kagal, Achille Fokoue, Paul T. Groth, Chris Biemann, Josiane Xavier Parreira, Lora Aroyo, Natasha F. Noy, Chris Welty, and Krzysztof Janowicz, editors, *The Semantic Web - ISWC 2013 - 12th International Semantic Web Conference, Sydney, NSW, Australia, October 21-25, 2013, Proceedings, Part I*, volume 8218 of *Lecture Notes in Computer Science*, pages 280–297. Springer, 2013.

[LPQPH11] Danh Le-Phuoc, Hoan Nguyen Mau Quoc, Josiane Xavier Parreira, and Manfred Hauswirth. The linked sensor middleware–connecting the real world and the semantic web. *Proceedings of the Semantic Web Challenge*, 152, 2011.

[LPT99] Ling Liu, Calton Pu, and Wei Tang. Continual queries for internet scale event-driven information delivery. *IEEE Trans. on Knowl. and Data Eng.*, 11(4):610–628, July 1999.

[LR86] Terry Landers and Ronni L Rosenberg. An overview of multibase. In *Distributed systems, Vol. II: distributed data base systems*, pages 391–421. Artech House, Inc., 1986.

[LSK14] Markus Lanthaler, Manu Sporny, and Gregg Kellogg. JSON-LD 1.0. W3C recommendation, W3C, January 2014. `http://www.w3.org/TR/2014/REC-json-ld-20140116/`.

[Luc01] David C. Luckham. *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*. Addison-Wesley Longman Publishing Co., Inc., 2001.

[LUQ14] Chang Liu, Jacopo Urbani, and Guilin Qi. Efficient RDF stream reasoning with graphics processing units (GPUs). In Chin-Wan Chung, Andrei Z. Broder, Kyuseok Shim, and Torsten Suel, editors, *23rd International World Wide Web Conference, WWW '14, Seoul, Republic of Korea, April 7-11, 2014, Companion Volume*, pages 343–344. ACM, 2014.

[Mar95] Gary Marchionini. A small matter of programming perspectives on end user computing, by bonnie nardie. *JASIS*, 46(1):78–79, 1995.

[MBH+13] Marco Miglierina, Marco Balduini, Narges Shahmandi Hoonejani, Elisabetta Di Nitto, and Danilo Ardagna. Exploiting stream reasoning to monitor multi-cloud applications. In *Proceedings of the 2nd International Workshop on Ordering and Reasoning, OrdRing 2013, Co-located with the 12th International Semantic Web Conference (ISWC 2013)*, pages 33–36. CEUR-WS, 2013.

[MCD+15] Andrea Mauri, Jean-Paul Calbimonte, Daniele Dell'Aglio, Marco Balduini, Emanuele Della Valle, and Karl Aberer. Where are the RDF streams?: On deploying RDF streams on the web of data with triplewave. In Serena Villata, Jeff Z. Pan, and Mauro Dragoni, editors, *Proceedings of the ISWC 2015 Posters & Demonstrations Track co-located with the 14th International Semantic Web Conference (ISWC-2015), Bethlehem, PA, USA, October 11, 2015.*, volume 1486 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2015.

[MFC+07] Deborah L. McGuinness, Peter Fox, Luca Cinquini, Patrick West, Jose Garcia, James L. Benedict, and Don Middleton. The virtual solar-terrestrial observatory: A deployed semantic web application case study for scientific research. In *Proceedings of the Twenty-Second AAAI Conference on Artificial Intelligence, July 22-26, 2007, Vancouver, British Columbia, Canada*, pages 1730–1737. AAAI Press, 2007.

[MFM11] Alexandra Moraru, Carolina Fortuna, and Dunja Mladenic. A system for publishing sensor data on the semantic web. *CIT*, 19(4):239–245, 2011.

[mic] Microsoft corporation: Microsoft StreamInsight. `https://technet.microsoft.com/en-us/library/ee362541%28v=sql.111%29.aspx`. accessed 16 June 2016.

[MIKS00] Eduardo Mena, Arantza Illarramendi, Vipul Kashyap, and Amit P. Sheth. OBSERVER: an approach for query processing in global information systems based on interoperation across pre-existing ontologies. *Distributed and Parallel Databases*, 8(2):223–271, 2000.

[MLAN11] Mohamed Morsey, Jens Lehmann, Sören Auer, and Axel-Cyrille Ngonga Ngomo. DBpedia SPARQL benchmark: Performance assessment with real queries on real data. In *Proceedings of the 10th International Conference on The Semantic Web - Volume Part I*, ISWC'11, pages 454–469, Berlin, Heidelberg, 2011. Springer-Verlag.

[MRM+10] Zaki Malik, Abdelmounaam Rezgui, Brahim Medjahed, Mourad Ouzzani, and A. Krishna Sinha. Semantic integration in geosciences. *Int. J. Semantic Computing*, 4(3):301–330, 2010.

[MRS+07] Zaki Malik, Abdelmounaam Rezgui, A. Krishna Sinha, Kai Lin, and Athman Bouguettaya. DIA: A web services-based infrastructure for semantic integration in geoinformatics. In *2007 IEEE International Conference on Web Services (ICWS 2007), July 9-13, 2007, Salt Lake City, Utah, USA*, pages 1016–1023. IEEE Computer Society, 2007.

[MUHB14] Alessandro Margara, Jacopo Urbani, Frank van Harmelen, and Henri Bal. Streaming the web: Reasoning over dynamic data. *Web Semantics: Science, Services and Agents on the World Wide Web*, 25(0), 2014.

[MWA+03] Rajeev Motwani, Jennifer Widom, Arvind Arasu, Brian Babcock, Shivnath Babu, Mayur Datar, Gurmeet Singh Manku, Chris Olston, Justin Rosenstein, and Rohit Varma. Query processing, approximation, and resource management in a data stream management system. In *CIDR*, 2003.

[NCSP10] Anne H. H. Ngu, Michael P. Carlson, Quan Z. Sheng, and Hye-young Paik. Semantic-based mashup of composite applications. *IEEE Trans. Serv. Comput.*, 3(1):2–15, 2010.

[NM01] Natalya F Noy and Deborah L McGuinness. *Ontology development 101: A guide to creating your first ontology.* Stanford knowledge systems laboratory technical report KSL-01-05 and Stanford medical informatics technical report SMI-2001-0880, 2001.

[Noy04] Natalya Fridman Noy. Semantic integration: A survey of ontology-based approaches. *SIGMOD Record*, 33(4):65–70, 2004.

[oFO88] AME Study Group on Functional Organization. Organizational Renewal—Tearing down the functional silos. *AME Target*, 1988.

[Pal06] Michael Palmer. Data is the new oil. `http://ana.blogs.com/maestros/2006/11/data_is_the_new.html`, 2006.

[PB02] Hardy Pundt and Yaser Bishr. Domain ontologies for data sharing-an example from environmental monitoring using field GIS. *Comput. Geosci.*, 28(1):95–102, February 2002.

[PHS10] Harshal Patni, Cory Henson, and Amit Sheth. Linked sensor data. In *Collaborative Technologies and Systems (CTS), 2010 International Symposium on*, pages 362–370. IEEE, 2010.

[PRM11] Stefan Pietschmann, Carsten Radeck, and Klaus Meißner. Semantics-based discovery, selection and mediation for presentation-oriented mashups. In *Proceedings of the 5th International Workshop on Web APIs and Service Mashups*, Mashups '11, pages 7:1–7:8, New York, NY, USA, 2011. ACM.

[PS08] Eric Prud'hommeaux and Andy Seaborne. SPARQL query language for RDF. W3C recommendation, W3C, January 2008. http://www.w3.org/TR/2008/REC-rdf-sparql-query-20080115/.

[PST04] Helena Sofia Pinto, Steffen Staab, and Christoph Tempich. DILIGENT: towards a fine-grained methodology for distributed, loosely-controlled and evolving engineering of ontologies. In Ramon López de Mántaras and Lorenza Saitta, editors, *Proceedings of the 16th Eureopean Conference on Artificial Intelligence, ECAI'2004, including Prestigious Applicants of Intelligent Systems, PAIS 2004, Valencia, Spain, August 22-27, 2004*, pages 393–397. IOS Press, 2004.

[PTRC07] Ken Peffers, Tuure Tuunanen, Marcus A Rothenberger, and Samir Chatterjee. A design science research methodology for information systems research. *Journal of management information systems*, 24(3):45–77, 2007.

[PZC+13] Charith Perera, Arkady B. Zaslavsky, Peter Christen, Michael Compton, and Dimitrios Georgakopoulos. Context-aware sensor search, selection and ranking model for internet of things middleware. In *2013 IEEE 14th International Conference on Mobile Data Management, Milan, Italy, June 3-6, 2013 - Volume 1*, pages 314–322. IEEE Computer Society, 2013.

[QSLPH12] Hoan Nguyen Mau Quoc, Martin Serrano, Danh Le-Phuoc, and Manfred Hauswirth. Super stream collider – Linked stream mashups for everyone. *Proceedings of the Semantic Web Challenge co-located with ISWC 2012*, 2012.

[RFH10] Maria Rüther, Joachim Fock, and Joachim Hübener. Linked environment data. In Klaus Greve and Armin B. Cremers, editors, *EnviroInfo 2010: Integration of Environmental Information in Europe, Proceedings of the 24th International Conference on Informatics for Environmental Protection, Cologne/Bonn, Germany*, pages 470–479. Shaker Verlag, Aachen, 2010.

[RNT12] Mikko Rinne, Esko Nuutila, and Seppo Törmä. INSTANS: High-Performance event processing with standard RDF and SPARQL. In Birte Glimm and David Huynh, editors, *International Semantic Web Conference (Posters & Demos)*, volume 914 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2012.

[RPZ10] Yuan Ren, Jeff Z. Pan, and Yuting Zhao. Towards soundness preserving approximation for ABox reasoning of OWL2. In *Proceedings of the 4th International Workshop on Ontology Dynamics (IWOD) 2010*, 2010.

[RZB+12] Bernd Resch, Alexander Zipf, Euro Beinat, Philipp Breuss-Schneeweis, and Marc Boher. Towards the live city–paving the way to real-time urbanism. *International Journal on Advances in Intelligent Systems*, 5(3 and 4):470–482, 2012.

[SFGPFL12] Mari Carmen Suárez-Figueroa, Asunción Gómez-Pérez, and Mariano Fernández-López. *Ontology Engineering in a Networked World*, chapter The NeOn Methodology for Ontology Engineering, pages 9–34. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[SGH+11] Michael Schmidt, Olaf Görlitz, Peter Haase, Günter Ladwig, Andreas Schwarte, and Thanh Tran. FedBench: A benchmark suite for federated semantic data query processing. In *The Semantic Web - ISWC 2011 - 10th ISWC, Bonn, Germany, October 23-27, 2011, Proceedings, Part I*, pages 585–600, 2011.

[SRK14] Markus Stocker, Mauno Rönkkö, and Mikko Kolehmainen. Towards an ontology for situation assessment in environmental monitoring. In *Proceedings of the 7th International Congress on Environmental Modelling and Software*, pages 1281–1288, San Diego, CA, USA, 2014.

[SSS04] York Sure, Steffen Staab, and Rudi Studer. On-to-knowledge methodology (OTKM). In Steffen Staab and Rudi Studer, editors, *Handbook on Ontologies*, International Handbooks on Information Systems, pages 117–132. Springer, 2004.

[SUM+13] Thomas Scharrenbach, Jacopo Urbani, Alessandro Margara, Emanuele Della Valle, and Abraham Bernstein. Seven commandments for benchmarking semantic flow processing systems. In *The Semantic Web: Semantics and Big Data*, pages 305–319. Springer, 2013.

[TAM13] Tatiana Tarasova, Massimo Argenti, and Maarten Marx. Semantically-Enabled environmental data discovery and integration: Demonstration using the iceland volcano use case. In *Knowledge Engineering and the Semantic Web*, page 289–297. Springer, Berlin, Heidelberg, 2013.

[TAR10] Kerry Taylor, Arun Ayyagari, and David De Roure, editors. *Proceedings of the 3rd International Workshop on Semantic Sensor Networks, SSN 2010, Shanghai, China, November 7, 2010*, volume 668 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2010.

[Tar13] Tatiana Tarasova. ENVRI project deilverable 4.2: Semantic component. 2013.

[TDW+13] Tuan-Dat Trinh, Ba-Lam Do, Peter Wetz, Amin Anjomshoaa, and A Min Tjoa. Linked Widgets - An approach to exploit open government data. In *Proceedings of the 15th International Conference on Information Integration and Web-based Applications & Services*, page 438–442. ACM, 2013.

[tib] TIBCO software: Tibco BusinessEvents. `http://www.tibco.com/products/event-processing/complex-event-processing/businessevents`. accessed 16 June 2016.

[TKSA12] Mohsen Taheriyan, Craig A. Knoblock, Pedro Szekely, and José Luis Ambite. Rapidly integrating services into the linked data cloud. In *Proceedings of the 11th International Conference on The Semantic Web - Volume Part I*, Lecture Notes in Computer Science, pages 559–574. Springer Berlin Heidelberg, 2012.

[TKSA13] Mohsen Taheriyan, Craig A. Knoblock, Pedro Szekely, and José Luis Ambite. A graph-based approach to learn semantic descriptions of data sources. In Harith Alani, Lalana Kagal, Achille Fokoue, Paul Groth, Chris Biemann, JosianeXavier Parreira, Lora Aroyo, Natasha Noy, Chris Welty, and Krzysztof Janowicz, editors, *The Semantic Web – ISWC 2013*, volume 8218 of *Lecture Notes in Computer Science*, pages 607–623. Springer Berlin Heidelberg, 2013.

[Tri16] Tuan-Dat Trinh. *Mashup-based Linked Data Integration*. PhD thesis, TU Wien, 2 2016.

[TWD+14a] Tuan-Dat Trinh, Peter Wetz, Ba-Lam Do, Amin Anjomshoaa, Elmar Kiesling, and A Min Tjoa. Linked Widgets Platform: Lowering the barrier for open data exploration. In Valentina Presutti, Eva Blomqvist, Raphaël Troncy, Harald Sack, Ioannis Papadakis, and Anna Tordai, editors, *The Semantic Web: ESWC 2014 Satellite Events - ESWC 2014 Satellite Events, Anissaras, Crete, Greece, May 25-29, 2014, Revised Selected Papers*, volume 8798 of *Lecture Notes in Computer Science*, pages 171–182. Springer, 2014.

[TWD+14b] Tuan-Dat Trinh, Peter Wetz, Ba-Lam Do, Amin Anjomshoaa, Elmar Kiesling, and A Min Tjoa. Open linked widgets mashup platform. In *Proceedings of the AI Mashup Challenge 2014 (ESWC Satellite Event)*, page 9. CEUR Workshop Proceedings, 2014.

[TWD+14c] Tuan-Dat Trinh, Peter Wetz, Ba-Lam Do, Amin Anjomshoaa, Elmar Kiesling, and A Min Tjoa. A web-based platform for dynamic integration of heterogeneous data. In *Proceedings of the 16th International Conference on Information Integration and Web-based Applications & Services*, iiWAS '14, pages 253–261, New York, NY, USA, 2014. ACM.

[TWD+15] Tuan-Dat Trinh, Peter Wetz, Ba-Lam Do, Elmar Kiesling, and A Min Tjoa. Distributed mashups: a collaborative approach to data integration. *IJWIS*, 11(3):370–396, 2015.

[UG04] Michael Uschold and Michael Gruninger. Ontologies and semantics for seamless connectivity. *ACM SIGMod Record*, 33(4):58–64, 2004.

[Ull97] Jeffrey D Ullman. Information integration using logical views. In *Database Theory—ICDT'97*, page 19–40. Springer, Berlin, Heidelberg, 1997.

[UMD+13] Jacopo Urbani, Jason Maassen, Niels Drost, Frank J. Seinstra, and Henri E. Bal. Scalable RDF data compression with MapReduce. *Concurrency and Computation: Practice and Experience*, 25(1):24–39, 2013.

[Umw11] Wissenschaftlicher Beirat Globale Umweltveränderungen. *World in transition: a social contract for sustainability*. WBGU, Berlin, 2011.

[UN12] UN. World urbanization prospects - the 2011 revision. Technical report, United Nations Department of Economic and Social Affairs, 2012.

[UNE12a] UNEP. GEO-5 assessment full report. `http://www.unep.org/geo/pdfs/geo5/GEO5_report_full_en.pdf`, 2012.

[UNE12b] UNEP. GEO-5 report press release. Technical report, United Nations Environment Programme, 2012.

[UNE16] UNEP. UNEP 2015 annual report. `http://apps.unep.org/publications/index.php?option=com_pub&task=download&file=012048_en`, 2016.

[Uni03] European Union. Directive on public access to environmental information and repealing council directive, January 2003.

[UNP15] Jürgen Umbrich, Sebastian Neumaier, and Axel Polleres. Quality assessment and evolution of open data portals. In *3rd International Conference on Future Internet of Things and Cloud, FiCloud 2015, Rome, Italy, August 24-26, 2015*, pages 404–411. IEEE, 2015.

[vAMPR04] R Hevner von Alan, Salvatore T March, Jinsoo Park, and Sudha Ram. Design science in information systems research. *MIS quarterly*, 28(1):75–105, 2004.

[VCHF09] Emanuele Della Valle, Stefano Ceri, Frank van Harmelen, and Dieter Fensel. It's a streaming world! reasoning upon rapidly changing information. *IEEE Intelligent Systems*, 24(6):83–89, 2009.

[VSVD+11] Ruben Verborgh, Thomas Steiner, Davy Van Deursen, Rik Van de Walle, and Joaquim Gabarró Vallés. Efficient Runtime Service Discovery and Consumption with Hyperlinked RESTdesc. In *Proceedings of the 7th International Conference on Next Generation Web Services Practices*, pages 373–379, 2011.

[WAT13] Peter Wetz, Amin Anjomshoaa, and A Min Tjoa. A survey on environmental open data in austria. In *IEEE International Conference on Systems, Man, and Cybernetics, Manchester, SMC 2013, United Kingdom, October 13-16, 2013*, pages 4566–4570. IEEE, 2013.

[WHO16] WHO. Preventing disease through healthy environments: a global assessment of the burden of disease from environmental risks. http://apps.who.int/iris/bitstream/10665/204585/1/9789241565196_eng.pdf, 2016.

[WTD+13] Peter Wetz, Tuan-Dat Trinh, Ba-Lam Do, Amin Anjomshoaa, and A Min Tjoa. Austrian environmental data consumption — a mashup-based approach. In *Proceedings of the 1st International Workshop on Semantic Machine Learning and Linked Open Data (SML2OD) for Agricultural and Environmental Informatics co-located with the 12th International Semantic Web Conference (ISWC 2013), Sydney, Australia, 2013*, 2013.

[WTD+14] Peter Wetz, Tuan-Dat Trinh, Ba-Lam Do, Amin Anjomshoaa, Elmar Kiesling, and A Min Tjoa. Towards an environmental information system for semantic stream data. In Jorge Marx Gómez, Michael Sonnenschein, Ute Vogel, Andreas Winter, Barbara Rapp, and Nils Giesen, editors, *28th International Conference on Informatics for Environmental Protection: ICT for Energy Effieciency, EnviroInfo 2014, Oldenburg, Germany, September 10-12, 2014.*, pages 637–644. BIS-Verlag, 2014.

[WTD+16] Peter Wetz, Tuan-Dat Trinh, Ba-Lam Do, Amin Anjomshoaa, Elmar Kiesling, and A Min Tjoa. Towards an environmental decision-making system: A vocabulary to enrich stream data. In Jorge Marx Gomez, Michael Sonnenschein, Ute Vogel, Andreas Winter, Barbara Rapp, and Nils Giesen, editors, *Advances and New Trends in Environmental and Energy Informatics*, Progress in IS, pages 317–335. Springer International Publishing, 2016.

[WVV+01] Holger Wache, Thomas Vögele, Ubbo Visser, Heiner Stuckenschmidt, Gerhard Schuster, Holger Neumann, and Sebastian Hübner. Ontology-based integration of information - a survey of existing approaches. In *IJCAI-01 workshop: ontologies and information sharing*, pages 108–117, 2001.

[ZD04] Patrick Ziegler and Klaus R. Dittrich. *Building the Information Society: IFIP 18th World Computer Congress Topical Sessions 22–27 August 2004 Toulouse, France*, chapter Three Decades of Data Intecration — all Problems Solved?, pages 3–12. Springer US, Boston, MA, 2004.

[ZDCC12] Ying Zhang, Pham Minh Duc, Óscar Corcho, and Jean-Paul Calbimonte. SRBench: a streaming RDF/SPARQL benchmark. In *The Semantic Web - ISWC 2012 - 11th ISWC, Boston, MA, USA, November 11-15, 2012, Proceedings, Part I*, pages 641–657. Springer, 2012.

Curriculum Vitae

Address	DI (FH) Peter Wetz Schumanngasse 3 / 7 1180 Wien AUSTRIA
Date of birth	July 12th, 1986

Education

2013–2016
Continuation of PhD Studies in Engineering Science
Doctoral College Environmental Informatics
TU Wien

2011–2012
PhD Studies in Engineering Science
Fulfillment of course requirements
Graz University of Technology

2005–2010
Diploma in Information Managment
FH JOANNEUM, Graz

Job Experience

2013–2016
Collegiate at the Doctoral College Environmental Informatics
TU Wien

2010–2013
Junior Researcher
Know-Center competence centre for knowledge-based applications and systems, Graz

2008
Systems Engineer Intern
Graz AG

2007
Software Development Intern
Mondi Frantschach GmbH

Selected Publications

Wetz, P., Anjomshoaa, A., Tjoa, A M. (2013), A Survey on Environmental Open Data in Austria. In *Proceedings of the 2013 IEEE International Conference on Systems, Man, and Cybernetics (SMC2013)*, IEEE, pp. 4566 – 4570.

Wetz, P., Trinh, T. D., Do, B. L., Anjomshoaa, A., Tjoa, A M. (2013), Austrian Environmental Data Consumption—A Mashup-based Approach. In *Proceedings of the 1st International Workshop on Semantic Machine Learning and Linked Open Data (SML2OD) for Agricultural and Environmental Informatics co-located with the 12th International Semantic Web Conference (ISWC 2013)*.

Wetz, P., Trinh, T. D., Do, B. L., Anjomshoaa, A., Kiesling, E., Tjoa, A M. (2014), Towards an Environmental Information System for Semantic Stream Data. In P*roceedings of the 28th Conference on Environmental Informatics – Informatics for Environmental Protection, Sustainable Development and Risk Management (EnviroInfo 2014)*, BIS-Verlag, Oldenburg, pp. 637 – 644. **Best Paper Award**

Wetz, P., Tuan, D. T., Lam, D. B., Anjomshoaa, A., Kiesling, E., Tjoa, A M. (2015), Towards an Environmental Decision-Making System: A Vocabulary to Enrich Stream Data. In *Advances and New Trends in Environmental and Energy Informatics*, Springer, ISSN: 2196-8705, pp. 317-335.

Kolchin, M., Wetz, P. (2015), Demo: YABench–Yet Another RDF Stream Processing Benchmark. In *RDF Stream Processing Workshop co-located with the 11th Extended Semantic Web Conference (ESWC 2014)*, Portoroz, Slovenia.

Kolchin, M., Wetz, P., Kiesling, E., Tjoa, A M. (2016), YABench: A comprehensive framework for RDF stream processor correctness and performance assessment. In *Web Engineering–16th International Conference on Web Engineering*, ICWE 2016, Lugano, Switzerland, June 6-9, 2016, Proceedings, Springer Berlin Heidelberg.

Grants & Awards

2014
Best Paper Award EnviroInfo Conference
Student Grant ISWC
Winner of the Web Intelligence Summer School Hackathon
Student Grant Web Intelligence Summer School
Second place "AI Mashup Challenge" at ESWC

2012
Second Place in "Poster & Demo Challenge" at I-KNOW

2005–2007
Scholarship for academic excellence at FH JOANNEUM