

# Ontology-based Access to Sensor Data Streams



**Jean-Paul Calbimonte**

Facultad de Informática

Universidad Politécnica de Madrid

*PhD Thesis*

Supervisor: Oscar Corcho

January 2013



## Acknowledgements

A big thanks to Oscar for guiding me through the good, the bad and the ugly in the research world. A huge one also for Asun, who gave me the opportunity to come to Madrid and encouraged me to keep calm and enjoy our work during all these years. It has really been a pleasure to work with both of you.

Thanks to all the people at the Ontology Engineering Group (OEG), I really enjoyed being there, including the tapas workshops, football matches, conference and project travels, cafeteria lunch, the heat, Thursday talks, and everything that is inherent to an OEGer. Cheers to Boris, Freddy, Raúl, Mora, Andrés, Víctor, María, Elena, Luis, Filip, Dani, Idafen, Migue, Esther, Ghis, Diego, Alcázar, Dani, Ana, Mari-Carmen, Jorge, Jose Angel, and all those who came and went.

A special thanks to Alasdair for the work we did in Manchester, it was essential to get this thing through. Also to Hoyoung for all the support during my time at EPFL, which helped shaping many aspects of this thesis. I would like to make this extensive to all the colleagues I worked with during those times: Zhixian, Karl, Ixent, Alvaro and Norman.

I must say thanks to Fabio as well, who first introduced me to ontologies and semantics, and to the research world altogether, back there in Lausanne.

I would also like to acknowledge the support of the European FP7 Projects SemSor-Grid4Env, PlanetData, and the Innpronta Ciudad2020 project. Most of the work in this thesis was performed in the context of these projects, or applied to use cases in them. Thanks to the people involved in all of that: Jennie, Duc, Edna, Kostis, Manos, Kevin, Jason, Alex, Juanjo, Manolis and all.

To all those I did not mention, my friends at home, call it Spain, Bolivia or Switzerland, thanks to you too!

None of this would have been possible without the support and love of my fiancée, my mom and dad, brothers and sisters. Big hug to all of you, spread all over the globe.

## Abstract

Sensor networks are increasingly becoming one of the main sources of Big Data on the Web. However, the observations that they produce are made available with heterogeneous schemas, vocabularies and data formats, making it difficult to share and reuse these data for other purposes than those for which they were originally set up. In this thesis we address these challenges, considering how we can transform streaming raw data to rich ontology-based information that is accessible through continuous queries for streaming data. Our main contribution is an ontology-based approach for providing data access and query capabilities to streaming data sources, allowing users to express their needs at a conceptual level, independent of implementation and language-specific details. We introduce novel query rewriting and data translation techniques that rely on mapping definitions relating streaming data models to ontological concepts. Specific contributions include:

- The syntax and semantics of the SPARQL<sub>Stream</sub> query language for ontology-based data access, and a query rewriting approach for transforming SPARQL<sub>Stream</sub> queries into streaming algebra expressions.
- The design of an ontology-based streaming data access engine that can internally reuse an existing data stream engine, complex event processor or sensor middleware, using R2RML mappings for defining relationships between streaming data models and ontology concepts.

Concerning the sensor metadata of such streaming data sources, we have investigated how we can use raw measurements to characterize streaming data, producing enriched data descriptions in terms of ontological models. Our specific contributions are:

- A representation of sensor data time series that captures gradient information that is useful to characterize types of sensor data.
- A method for classifying sensor data time series and determining the type of data, using data mining techniques, and a method for extracting semantic sensor metadata features from the time series.

---

# Contents

<b>Contents</b>	<b>v</b>
<b>List of Figures</b>	<b>xi</b>
<b>List of Tables</b>	<b>xv</b>
<b>Nomenclature</b>	<b>xv</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Dissemination of Results . . . . .	4
<b>2 State of the Art</b>	<b>7</b>
2.1 Streaming Data Querying and Processing . . . . .	8
2.1.1 Data Stream and Event Processing Models . . . . .	8
2.1.2 Continuous Query Languages . . . . .	15
2.1.3 System Implementations . . . . .	19
2.2 Ontology-based Access to Relational Data Sources . . . . .	26
2.2.1 Data Generation and Querying . . . . .	28
2.2.2 Mappings for RDB2RDF . . . . .	29
2.2.3 RDB2RDF Implementations . . . . .	32
2.3 Extensions for RDF Stream Querying . . . . .	34
2.3.1 SPARQL Extensions for Streams . . . . .	34
2.3.2 Publishing Data Streams as Linked Data . . . . .	37
2.4 Sensor Ontology Modeling . . . . .	38
2.4.1 Ontologies for Sensor Data and Metadata . . . . .	39
2.4.2 The SSN Ontology . . . . .	40
2.5 Sensor Data Approximation and Classification . . . . .	43
2.5.1 Data Approximation and Compression . . . . .	43

2.5.2	Time Series Classification . . . . .	47
2.5.3	Characterizing Unknown Sensor Data . . . . .	49
2.6	Discussion . . . . .	50
<b>3</b>	<b>Hypotheses and Contributions</b>	<b>53</b>
3.1	Problem Statement . . . . .	53
3.2	Hypotheses . . . . .	55
3.2.1	Assumptions . . . . .	55
3.2.2	Limitations . . . . .	56
3.3	Contributions . . . . .	57
<b>4</b>	<b>A SPARQL Extension for Querying Data Streams</b>	<b>59</b>
4.1	Features of a Semantically-enabled Streaming Query Language . . . . .	60
4.1.1	Discussion . . . . .	62
4.2	RDF Stream Model . . . . .	63
4.2.1	RDF triples and graphs . . . . .	63
4.2.2	RDF Stream Graphs . . . . .	63
4.3	SPARQL <sub>Stream</sub> Syntax . . . . .	64
4.3.1	SPARQL Syntax . . . . .	65
4.3.2	SPARQL <sub>Stream</sub> Extensions . . . . .	66
4.3.3	Comparison of SPARQL <sub>Stream</sub> with other Languages . . . . .	69
4.4	SPARQL <sub>Stream</sub> Semantics . . . . .	70
4.4.1	SPARQL Semantics . . . . .	70
4.4.2	SPARQL <sub>Stream</sub> Extensions Semantics . . . . .	71
4.4.3	Comparison of SPARQL <sub>Stream</sub> Semantics with other Languages . . . . .	72
4.5	Rewriting SPARQL <sub>Stream</sub> Queries . . . . .	73
4.5.1	Relational to RDF Mappings . . . . .	73
4.5.2	Definition of Mappings for Query Rewriting . . . . .	74
4.5.3	Rewriting to Algebra Expressions . . . . .	76
4.5.4	Rewriting Window Operators . . . . .	78
4.5.5	Rewriting other Operators . . . . .	79
4.6	Query Optimization . . . . .	80
4.7	Query Results Translation . . . . .	82
4.7.1	Projected Expressions . . . . .	83
4.7.2	Construct and Ask . . . . .	83
4.8	Discussion . . . . .	83

<b>5</b>	<b>Querying Semantic Sensor Networks</b>	<b>85</b>
5.1	Design of an Ontology-based Stream Query Processor . . . . .	86
5.1.1	Ontologies for describing Streaming Data . . . . .	86
5.1.2	Declarative Mappings from Streams to Ontologies . . . . .	87
5.1.3	Querying in terms of Ontologies . . . . .	87
5.1.4	Rewriting queries to native streaming languages . . . . .	87
5.1.5	Delegation of Query Execution . . . . .	88
5.1.6	Query Results Delivery . . . . .	89
5.2	Mapping Streams to Ontologies . . . . .	89
5.3	Ontology-based Streaming Data Query Architecture . . . . .	92
5.4	Query Rewriting Process . . . . .	94
5.4.1	Rewriting to Algebra Expressions . . . . .	94
5.4.2	Query Optimization . . . . .	95
5.5	Query Instantiation and Delegation . . . . .	95
5.5.1	DSMS Query processing . . . . .	96
5.5.2	Complex Event Processing . . . . .	96
5.5.3	Middleware Query Processing . . . . .	97
5.6	Query Evaluation . . . . .	97
5.6.1	Snapshot Queries . . . . .	98
5.6.2	Continuous Queries . . . . .	98
5.6.3	Pull-based Data Retrieval . . . . .	99
5.6.4	Push-based Data Retrieval . . . . .	99
5.7	Implementing Ontology-based Streaming Query Rewriting . . . . .	99
5.7.1	Query Processing . . . . .	100
5.8	Discussion . . . . .	104
<b>6</b>	<b>Sensor Metadata and Data Characterization</b>	<b>107</b>
6.1	From Raw Measurements to Semantic Metadata . . . . .	108
6.2	Representing Sensor Data with Segment Slopes . . . . .	111
6.2.1	Background: Piecewise Linear Representation . . . . .	111
6.2.2	Slope Distributions . . . . .	112
6.2.3	Choosing the angle divisions . . . . .	113
6.3	Deriving Semantic Metadata . . . . .	114
6.3.1	Semantic Descriptions . . . . .	114
6.3.2	Data Classification . . . . .	115
6.3.3	Using Partial Data Subsets . . . . .	116



6.3.4	Querying using the Analysis Results . . . . .	116
6.4	Discussion . . . . .	117
<b>7</b>	<b>Experimentation with SPARQL<sub>Stream</sub></b>	<b>119</b>
7.1	SemSorGrid4Env . . . . .	119
7.1.1	Semantic Integrator . . . . .	121
7.1.2	Integration and Query Service . . . . .	122
7.1.3	Use-case: Coastal Sensors for Flood Warning . . . . .	123
7.1.4	Rewriting Examples for Coastal Sea Sensors . . . . .	125
7.2	Swiss Experiment Metadata and Querying . . . . .	129
7.2.1	Modeling Sensor Data with the SSN Ontology . . . . .	130
7.2.2	Querying GSN with SPARQL <sub>Stream</sub> . . . . .	132
7.3	Ciudad2020 Bike Sharing . . . . .	136
7.3.1	Bike Sharing Modeling . . . . .	136
7.3.2	Querying Bike Observations . . . . .	138
7.3.3	Exploitation . . . . .	139
7.4	Discussion . . . . .	140
<b>8</b>	<b>Evaluation</b>	<b>143</b>
8.1	SPARQL <sub>Stream</sub> Evaluation . . . . .	143
8.1.1	Comparing the Query Execution w/wo Rewriting . . . . .	144
8.1.2	Evaluation of SPARQL <sub>Stream</sub> Performance . . . . .	146
8.1.3	SPARQL <sub>Stream</sub> Evaluation Discussion . . . . .	152
8.2	Comparison of the SPARQL <sub>Stream</sub> Expressiveness with other Languages . . . . .	152
8.2.1	DSMS: SNEEqL and CQL . . . . .	152
8.2.2	CEP: Esper . . . . .	154
8.2.3	Middleware APIs: GSN and Cosm . . . . .	155
8.2.4	Expressiveness Discussion . . . . .	155
8.3	Functional Evaluation of SPARQL <sub>Stream</sub> . . . . .	156
8.3.1	SRBench Queries . . . . .	158
8.3.2	Functional Evaluation . . . . .	159
8.4	Evaluation of the Sensor Data Characterization . . . . .	161
8.4.1	Classification in Swiss Experiment and AEMET . . . . .	162
8.4.2	Classification with Partial Information . . . . .	165
8.4.3	Comparison with SAX Classification . . . . .	165
8.4.4	Discussion of the Characterization Evaluation . . . . .	167
8.5	Evaluation Conclusions . . . . .	168

<b>9 Conclusions</b>	<b>171</b>
9.1 Review of Contributions . . . . .	172
9.2 Future Work . . . . .	174
9.3 Open Research Problems . . . . .	175
<b>References</b>	<b>179</b>



# List of Figures

2.1	Stream data model based on unbounded timestamped triples. . . . .	9
2.2	Example of sensor observations modelled as streaming timestamped tuples. . . . .	10
2.3	Common event relationships: precedence, causality and aggregation. . . . .	11
2.4	Complex event generation model. . . . .	12
2.5	One-off and continuous query processing models. . . . .	13
2.6	Windows $w_1$ and $w_2$ over a data stream. . . . .	14
2.7	SSN ontology main classes and modules. . . . .	41
2.8	Constant piecewise approximation. . . . .	45
2.9	Piecewise linear approximation. . . . .	45
4.1	Tree representation of the evaluation of a rewritten SPARQL <sub>Stream</sub> query . . . . .	79
4.2	Push down projection. . . . .	80
4.3	Push down projection in a union. . . . .	80
4.4	Push down selection. . . . .	81
4.5	Self-join simplification. . . . .	81
4.6	Empty join simplification. . . . .	82
4.7	Push down window. . . . .	82
4.8	Projected expressions in query results . . . . .	83
5.1	The SSN Ontology combined with domain ontologies . . . . .	90
5.2	Mapping from the wan7 sensor to a SSN ObservationValue. . . . .	91
5.3	Ontology-based sensor query rewriting, processing and data translation. . . . .	93
5.4	Rewriting modules: rewriting a SPARQL <sub>Stream</sub> query to algebra expression, and optimizing the expressions. . . . .	94
5.5	Query processing: query instantiation, one-off execution, pull and push data delivery. . . . .	98
5.6	Mapping from the wan7 sensor to a SSN Observation. . . . .	101

5.7	Algebra expresison after query rewriting. . . . .	101
6.1	An architecture for inferring semantic sensor metadata from raw measure- ments. . . . .	110
6.2	Piecewise linear approximation, construction of the convex hull. . . . .	111
6.3	Slopes symbolization and angle space division . . . . .	113
6.4	Distribution of the angles for the AEMET training set . . . . .	114
7.1	SemSorGrid4Env architecture components . . . . .	120
7.2	IQS Integration and Query Service operations interactions. . . . .	123
7.3	Algebra expression generated for the SPARQL <sub>Stream</sub> query in Listing 7.3. .	126
7.4	Algebra expression generated for the SPARQL <sub>Stream</sub> query in Listing 7.5. .	127
7.5	SSN Ontology combined with other vocabularies for representing sensor observations. . . . .	131
7.6	Swiss Experiment virtual sensor wannengrat_wan7. . . . .	133
7.7	Virtual sensor wannengrat_wan7 mapped to an Observation Value in the SSN Ontology. . . . .	134
7.8	Rewritten algebra expression in terms of the wannengrat_wan7 virtual sensor.	134
7.9	Rewritten algebra expression as a union of the wannengrat_wan4b and wannengrat_wan7 virtual sensors. . . . .	135
7.10	SSN based Bike Sharing Ontology . . . . .	137
7.11	The implemented approach for generating static RDF and querying RDF streams with SPARQL <sub>Stream</sub> from the citybik.es services. . . . .	139
8.1	Pull response times for different tuple rates and windows. . . . .	145
8.2	Push-delivery latency, without query/data translation. . . . .	146
8.3	Push-delivery latency, with query/data translation. . . . .	147
8.4	Response times for a single-sensor query, compared to a 26 sensor union query, for different tuple rates . . . . .	150
8.5	Response times for time window queries, with different slide parameters .	151
8.6	Response times for different number of queries launched simultaneously, for different tuple rates . . . . .	151
8.7	Swiss Experiment classification confusion matrix . . . . .	163
8.8	AEMET classification confusion matrix . . . . .	164
8.9	Precision in AEMET, not differencing the specific types wind speed (max) and wind direction (max). . . . .	164

8.10 AEMET Classification precision, for different partial datasets, in terms of the days of data used. . . . .	165
8.11 Swiss Experiment Classification precision, for different partial datasets, in terms of the days of data used. . . . .	166
8.12 Swiss Experiment Classification precision with SAX and the Slope representation. . . . .	166
8.13 AEMET Classification precision with SAX and the Slope representation. .	167



# List of Tables

2.1	Comparative table of streaming data/event query languages. . . . .	16
2.2	Comparative tables of DSMS, CEP and streaming sensor middleware. . . .	27
4.1	Comparative table of SPARQL <sub>Stream</sub> and other SPARQL extensions for streams.	69
5.1	Comparative table of SPARQL <sub>Stream</sub> ontology-based system and alternative approaches. . . . .	105
8.1	Comparison of SPARQL <sub>Stream</sub> expressiveness with underlying streaming processor systems. . . . .	156
8.2	Addressed features per query in SRBench . . . . .	160
8.3	Functional evaluation results of SRBench . . . . .	160





# Chapter 1

## Introduction

Sensors produce massive amounts of data every second, all around the globe. Ranging from environmental and weather observations, to real-time patient health monitoring, the availability of these data streams is dramatically changing the way of conceiving data-oriented applications, providing the possibility of performing real-time analysis, tracking the evolution of data values over time, etc.

Many of these applications pose complex requirements regarding data management and query processing. For example, sensors can help studying and forecasting hurricanes, and help preventing natural disasters in vulnerable regions. Monitoring the barometric pressure at sea level, which is one of the causes for hurricanes, can be combined with other wind speed measurements and satellite imaging to better predict extreme weather conditions<sup>1</sup>. Even during a hurricane, storm-tide and gauge sensors can help measuring the water level, wind speed, direction and precipitation, leading to estimate floods in potentially endangered areas (Patni et al., 2010). Another example can be found in the health domain, where the industry is producing affordable devices that track caloric burn, blood glucose or heartbeat rates, among other indicators for early diagnosis or alert systems. The use of such sensors is not limited to monitoring for the elderly or impaired, but also for studying activity, metabolism and sleep patterns for any person (González-Valenzuela et al., 2011).

This type of data fits naturally with applications that store or publish it in the cloud, for the purpose of aggregating, comparing, sharing, and also for ubiquitous access. In fact, sensor data has been normally managed and sometimes published by large companies and public administrations, including weather information, financial stock market

---

<sup>1</sup>NASA hurricanes research: [http://www.nasa.gov/mission\\_pages/hurricanes](http://www.nasa.gov/mission_pages/hurricanes)

values, traffic monitoring, coastal environments conditions, etc. However, to these publishers we must now add a new type of data producers that include citizens and a broader spectrum of devices. Using cheap pluggable micro-controller boards based on platforms such as Arduino, communities of artists, geeks and virtually any person, are building prototypes and artifacts for an impressive number of tasks. For instance, the Citizen Sensor<sup>1</sup> initiative proposes open-source DIY devices for wearable and stationary air pollution monitoring, publishing the data generated by these devices on the web. Another example is the Air Quality Egg (AQE)<sup>2</sup>, a community-developed project, aiming at making citizens participate in measuring and sharing local air quality data and publish it online.

Smartphones and mobile communication devices are another source of live streaming data about the surrounding world. GPS, accelerators, microphones, cameras, and other instruments available in these devices can produce such type of data and have led to the emergence of concepts coined as *participatory sensing*. This paradigm builds from the fact that sensing individuals sharing their data can build a comprehensive body of knowledge at large-scale (Lane et al., 2010).

In this new context, a set of challenges surface for the research community: (i) The heterogeneity of devices, data structures, schemas, and formats of the data made available by these sensors, make it hard to make sense, reuse or integrate this type of data with other data sources; (ii) The highly dynamic and continuously changing nature of these data streams pose important challenges for today's query processing and data management technologies. In sum, the problems of data velocity and volume are inherent in these scenarios, added to the high variety of data at web-scale. These are precisely the 3Vs (Laney, 2001), which have been recently associated to the concept of Big Data.

In this thesis we address some of these challenges, considering how we can transform streaming raw data to rich ontology-based information that is accessible through continuous queries for streaming data. Ontologies have long been proposed as a way of providing explicit semantics for data, so as to facilitate reuse, integration or reasoning over it. Besides, the last decade has witnessed the emergence of work on continuous query and event processing, beyond traditional database systems, which were based on transactional and relational principles. Both visions need to be understood in order to propose techniques and methods that combine them in a coherent way. The use of struc-

---

<sup>1</sup><http://www.citizensensor.cc>

<sup>2</sup><http://airqualityegg.wikispaces.com>

---

tured semantics on the data, using ontologies for representing both streaming data and the metadata that describes it, is the cornerstone of the work presented in this thesis.

The scope of our work is centered around two main aspects, which we develop throughout this document, and that we can summarize as follows:

- (i) *How to enable ontology-based access and querying over existing streaming data sources?* Just as static or stored data already exists in the form of databases, XML files, web pages, etc., streams are also readily available on the Web, under formats, schemas and technologies that make them sometimes hard or unfeasible to be reused, understood or reasoned about, in web-scale. We investigate the extensions needed in semantic technologies and ontology-based querying to support this type of data, and how it can be linked or mapped to high-level ontological models that are representative of a specific domain.
- (ii) *How to represent, classify and enrich semantic sensor metadata?* The streaming data to be queried requires models that reflect the high-level concepts of the domain of interest. However, at web scale there is a high degree of heterogeneity and the data that describes data streams (i.e. the metadata) is often misleading, inconsistent or missing. We study sensor data classification and characterization, in order to produce enriched data descriptions in terms of ontological models.

The body of this thesis is structured as follows. In Chapter 2 we provide an account of the State of the Art in the different areas related to our work, or that are relevant as background knowledge, including streaming data processing, relational to ontology data access, RDF streaming processing, and sensor data classification. In Chapter 3 we formally introduce the hypotheses of this thesis, and our specific contributions. In Chapter 4 we present the SPARQL<sub>Stream</sub> language, its syntax, semantics and its representation as algebra expressions. Chapter 5 describes the process of query rewriting and instantiation of algebra expressions to queries for DSMS, CEP or middleware. We also introduce the use of R2RML mappings for stream-to-ontology relationships. The sensor metadata characterization from raw data is presented in Chapter 6. Chapter 7 describes different use cases where we applied the principles and technologies presented in the previous three chapters. The evaluation of our research hypotheses is detailed in Chapter 8. Finally we present the overall conclusions, ongoing work and future research directions in Chapter 9.

## 1.1 Dissemination of Results

The contributions presented in this thesis have been published, presented or reported in international journals, conferences, workshops and technical reports, as detailed below.

Our contributions in Chapter 4 were presented in:

- Calbimonte, J.-P.; Corcho, O. & Gray, A. J. G. **Enabling ontology-based access to streaming data sources**. In *Proc. 9th International Semantic Web Conference ISWC*, Shanghai, China, 2010. 96-111. ([Calbimonte et al., 2010b](#))
- Calbimonte, J.-P.; Corcho, O. & Gray, A. J. G. **Ontology-based Access to Streaming Data**. In *Poster Proceedings of the 7th Extended Semantic Web Conference ESWC*, Heraklion, Greece, 2010. ([Calbimonte et al., 2010a](#))

The main contributions of Chapter 5 and further advances in Chapter 4 and some of the evaluation results of Chapter 8 were published in:

- Calbimonte, J.-P.; Jeung, H.; Corcho, O. & Aberer, K. **Enabling Query Technologies for the Semantic Sensor Web**. *International Journal On Semantic Web and Information Systems (IJSWIS)*, IGI, 2012, 8. 43-63. ([Calbimonte et al., 2012a](#))

The contributions of Chapter 6, its relationship with Chapter 5 and the corresponding evaluations presented in Chapter 8 were presented in:

- Calbimonte, J.-P.; Jeung, H.; Corcho, O. & Aberer, K. **Semantic Sensor Data Search in a Large-Scale Federated Sensor Network**. In *Proc. 4th International Workshop on Semantic Sensor Networks at ISWC*, Bonn, Germany, 2011. 14-29. ([Calbimonte et al., 2011d](#))
- Calbimonte, J.-P.; Yan, Z.; Jeung, H.; Corcho, O. & Aberer, K. **Deriving Semantic Sensor Metadata from Raw Measurements**. In *Proc. of the 5th International Workshop on Semantic Sensor Networks SSN at ISWC*, Boston, USA, 2012. 33-48. ([Calbimonte et al., 2012b](#))
- Calbimonte, J.-P.; Jeung, H. & Corcho, O. **Querying Semantically Enriched Sensor Observations: Short Paper**. In *Proc. 6th International Workshop on Semantic Business Process Management at ESWC*, Heraklion, Greece, 2011. ([Calbimonte et al., 2011c](#))
- Corcho, O.; Priyatna, F.; Fortuna, C.; Grobelnik, M.; Calbimonte, J.-P.; García-Silva, A.; Jeung, H.; Novak, B. & Moraru, A. **Characterisation mechanisms for**

**unknown data sources.** Deliverable D1.1 Technical Report. PlanetData FP7, 2011. ([Corcho et al., 2011](#))

In addition, we are preparing a book chapter covering parts of Chapters 2, 4 and 5:

- Calbimonte, J.-P. & Corcho, O. **Evaluating SPARQL Queries over RDF Streams.** In *Linked Data Management: Principles and Techniques* (Ed. Harth, A., Hose, K., Schenkel, R.), CRC Press, 2013. (proposal accepted)

The results and use cases presented in Chapter 7 were published in:

- Gray, A.; García-Castro, R.; Kyzirakos, K.; Karpathiotakis, M.; Calbimonte, J.-P.; Page, K.; Sadler, J.; Frazer, A.; Galpin, I.; Fernandes, A. et al. **A semantically enabled service architecture for mashups over streaming and stored data.** In *Proceedings of the 8th Extended Semantic Web Conference ESWC*, Heraklion, Greece, 2011. 300-314. ([Gray et al., 2011a](#))
- Gray, A.; Sadler, J.; Kit, O.; Kyzirakos, K.; Karpathiotakis, M.; Calbimonte, J.-P.; Page, K.; García-Castro, R.; Frazer, A. & Galpin, I. et al. **A semantic sensor web for environmental decision support applications.** *Sensors*, MDPI, 2011, 11, 8855-8887. ([Gray et al., 2011b](#))
- Ruckhaus, E.; Calbimonte, J.-P.; García-Castro, R. & Corcho, O. **Short Paper: From Streaming Data to Linked Data—A Case Study with Bike Sharing Systems.** In *Proc. of the 5th International Workshop on Semantic Sensor Networks SSN at ISWC*, Boston, USA, 2012. 109-114. ([Ruckhaus et al., 2012](#))
- Calbimonte, J.-P.; Corcho, O. & Gray, A. J. G. **Implementation and Deployment of the SemSorGrid4Env ontology-based data integration service, Phase II.** Deliverable D4.2v2 Technical Report. SemSorGrid4Env FP7, 2011. ([Calbimonte et al., 2011a](#))
- Gray, A. J. G.; Galpin, I.; Fernandes, A. A. A.; Paton, N. W.; Page, K.; Sadler, J.; Kyzirakos, K.; Koubarakis, M.; Calbimonte, J.-P.; Corcho, O.; García-Castro, R.; Gabaldón, J. & Aparicio, J. **SemSorGrid4Env Architecture - Phase II.** Deliverable D1.3v2 Technical Report. SemSorGrid4Env FP7, 2010. ([Gray et al., 2010](#))

Finally, the remaining evaluation and benchmarking results of Chapter 8 were presented in:

- Zhang, Y.; Duc, P. M.; Corcho, O. & Calbimonte, J.-P. **SRBench: A Streaming RDF/SPARQL Benchmark**. In *Proceedings of the 11th International Semantic Web Conference ISWC*, Boston, USA, 2012, 641-657. ([Zhang et al., 2012a](#))
- Calbimonte, J.-P.; García-Castro, R.; Corcho, O. & Rodríguez, J. **Evaluation of the Ontology-based data integration service and the Ontologies**. Deliverable D4.4 Technical Report. SemSorGrid4Env FP7, 2011. ([Calbimonte et al., 2011b](#))
- Zhang, Y.; Duc, P. M.; Groffen, F.; Liarou, E.; Boncz, P.; Kersten, M.; Calbimonte, J.-P. & Corcho, O. **Benchmarking RDF Storage Engines**. Deliverable D1.2 Technical Report. PlanetData FP7, 2012. ([Zhang et al., 2012b](#))

## Chapter 2

# State of the Art

The work presented in this thesis focuses on exploiting streaming data sources, and proposes techniques that allow using semantically rich representations for querying them, as well as characterizing and enriching them. In this chapter we explore the different fields that serve as background for this work, and the approaches that have been proposed in the latest years related to the challenges that we address. Because our work deals with the problems of *querying observations* from streaming data sources and also with the *characterization* of data streams using semantic representations, we cover a wide range of related work in the different sections of this chapter, and we identify the existing limitations with respect to the challenges presented in Chapter 1.

First, we introduce the main concepts related to data streams and continuous query processing in Section 2.1. Then, we present in Section 2.2 the approaches related to representing and querying relational data in terms of ontologies, including the creation of relational-to-RDF mappings (also known as RDB2RDF mappings). These approaches are devised mainly for publishing static data sources in terms of semantically rich vocabularies. In Section 2.3 we describe the existing approaches that extend SPARQL with streaming operators, so as to support RDF stream querying. These three sections provide the background for ontology-based streaming data querying, which are complemented with the next two, dealing with their metadata and characterization. We focus on sensor data as a specific type of streaming data, and the different ontological representations used in the literature, in Section 2.4. Finally we discuss the existing work on time series representation and classification in Section 2.5.



## 2.1 Streaming Data Querying and Processing

Streaming data processing focuses on query and management of data streams, which can be seen as infinite and time-varying sequences of data values. Examples of data streams include stock market tickers, heart rate measurements or wave height observations from a coastal sensor network. These streams can also be abstracted as more complex events whose patterns can be detected or queried, and then delivered to a subscriber if they are relevant (Cugola and Margara, 2011). This type of applications differ from classical data management and query systems, and different approaches and solutions have been proposed in the latest years, establishing the research areas of streaming data and event processing.

The inherent characteristics of data streams pose challenges in the field of streaming query processing, which have been addressed in a number of previous works. We provide a brief summary of the *event and stream processing models*, *query languages* and event and streaming engine *system implementations* described in the literature.

### 2.1.1 Data Stream and Event Processing Models

The potentially infinite nature of streams and the need for continuous evaluation of data over time, are some of the fundamental reasons why managing streaming data differs significantly from classical static data. Database systems deal with mostly static and stored data, and their queries retrieve the state of the data at a given point in time, as opposed to monitoring the evolution of the data over time, in streaming and event data processing.

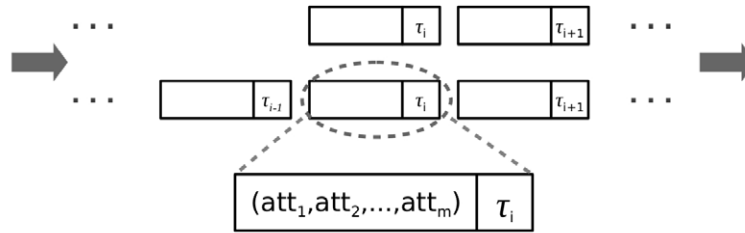
Highly dynamic data processing has been studied from different perspectives, which can be roughly classified into two main groups: Data Stream Processing and Complex Event Processing. In both cases, the type of data requirements differ greatly from traditional databases. For these streaming and event-based applications, normally the most recent values are more relevant than older ones. In fact, the focus of query processing is on the current observations and live-data, while historical data is often summarized, aggregated and stored for later analysis. Hence, data *recency* becomes a key factor in the design of these systems, which include time as a first-class citizen in their data models and processing algorithms.

Another central characteristic to this domain is the need for live and uninterrupted data processing and delivery. It is based on the principle of *continuous evaluation* of queries over the streams, in contrast with the existing models of stored relational databases (Babcock et al., 2002). Streaming values are pushed by the stream source (e.g. a sensor)

at possibly unknown rates and without explicit control of data arrival (Babu and Widom, 2001). Then, a query processor needs to continually monitor the arrival of tuples and check if they match one or more queries currently registered in the system.

In the following we describe some of the key aspects of streaming data and event models, continuous queries and recency-aware operators such as windows.

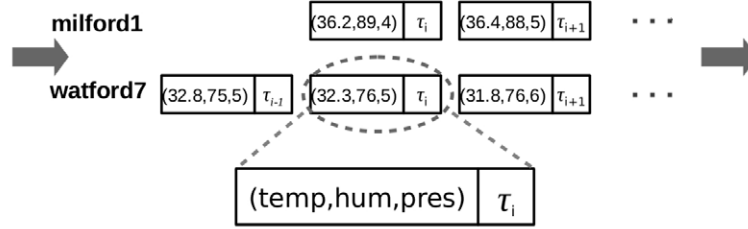
**Stream data model.** The field of streaming data processing emerged from relational database research, hence in the first proposals streams were seen merely as relations, in some cases limited to append-only updates as in Tapestry (Terry et al., 1992). Coming from the field of databases, the use of unbounded relational models to represent streams was a convenient decision, keeping query languages and operators intact, and only considering the new features as extensions. While the approach of Tapestry was restricted in terms of query expressiveness and limited to append-only tables, it already hinted at streams as sequences of data tuples arriving at a given order.



**Figure 2.1:** Stream data model based on unbounded timestamped triples.

A widely used data model perceives a stream as an unbounded sequence of tuples of values continuously appended (Golab and Özsu, 2003), each of which carries a time-based index, usually a sort of timestamp. The timestamp imposes a sequential order and typically indicates when the tuple has been produced, although this notion can be extended to consider the sampling, observation or arrival time. The tuple includes named attributes, such as the  $att_i$  in Figure 2.1 according to a schema, each having a given data type. Notice that for each non-decreasing timestamp  $\tau$  there may exist several tuples. Examples of systems following this idea include TelegraphCQ (Chandrasekaran et al., 2003), Aurora (Abadi et al., 2003) and STREAM (Arasu et al., 2007).

This model and variants of it have been used in several domains and use cases. As an example, environmental sensor observations can be abstracted as timestamped tuples, as depicted in Figure 2.2. In this case each stream is associated to a sensor location (e.g. *milford1*), and each tuple contains data about three observed properties: temperature,



**Figure 2.2:** Example of sensor observations modelled as streaming timestamped tuples.

humidity and air pressure, encoded as *temp*, *hum* and *pres* respectively. Alternative modelling decisions can be taken, such as using a single stream for all data, and adding an attribute to denote the location, etc.

A formal definition in these lines is provided in the STREAM system (Arasu et al., 2007). A stream  $s$  is defined as a set of elements of the form  $(t, \tau)$  where  $t$  is a tuple according to a given schema and  $\tau$  is a non-decreasing timestamp. This model can be modified, for instance, changing the tuple  $t$  for a complex object, using a DDL such as in Tribeca (Sullivan and Heybey, 1998) or abstract data types as in COUGAR (Bonnet et al., 2001). This model, complemented with normal relations, also covers alternatives such as chronicle models with persistent views that are periodically maintained, capturing only a summarization of the original data (Jagadish et al., 1995). The encoding format of each data stream element is independent of the abstract data model. Although most implementations use relational tuples as in a database table, with name-value attributes, others use XML for representing each tuple, e.g. NiagaraCQ (Chen et al., 2000).

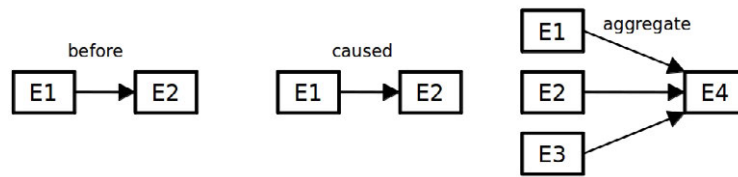
The notion of timestamp does not need to be necessarily related to the measured processing time: it is sometimes abstracted as a “tick” or integer-increasing value. Furthermore, the importance of this value resides in the ability of establishing an order among the stream tuples, hence it is a sequence and not a bag. The order is important as it allows discerning which tuples have already been processed against queries, and introduces the possibility of discarding older values, or creating bounded sub-sequences of data for processing. Nevertheless, even with timestamps, the tuples are not guaranteed to arrive in order, at least not in all processing models (Golab and Özsu, 2003), and therefore may require post-processing.

**Event data model.** A visible difference between the fields of data streams and event processing is on their data model. Contrary to table-like structures and schemas where the focus is on raw data management, events are complex abstractions of observations

or situations, modelled after a given domain of use (Cugola and Margara, 2011). Events are typically notified to subscribers if they are relevant to them (Chandy et al., 2011).

Event data management and query processing has been studied since the appearance of Active Object databases, where events are defined simply as “*something that happens at a point in time*” (Paton and Díaz, 1999). More specifically, events can be characterized by their *source*: e.g. generated by an operation, invocation, transaction, time-based, etc. Events may also have a *type*, primitive or composite, that serves to distinguish them and denote its semantics. For instance an event of type *TemperatureReading* represents a temperature observation, different from, e.g. a *HumidityReading*. Subscribers may receive notifications based on the types of events (type-based or topic-based subscription), but this is a limited approach, as events may have hierarchies, or produce complex and composite events.

Relations between events in terms of time and causality (simultaneousness, precedence, occurrence in an interval, etc.) are key for complex event query processing (Mei and Madden, 2009). In Figure 2.3 we illustrate common event relationships. In the first one event  $E1$  happens before  $E2$ . The events may be associated to a single point in time, but also to intervals, in which case an *overlap* relationship can also exist. The second relationship denotes causality,  $E1$  causes  $E2$ , which may also imply precedence, but the emphasis is on the origin of the resulting event. The last example denotes aggregation of several events ( $E1, E2, E3$ ) into a composite one  $E4$ . The nature of the aggregation is not specified in this case, but is meant to reflect the possibility of constructing complex events from simpler ones.

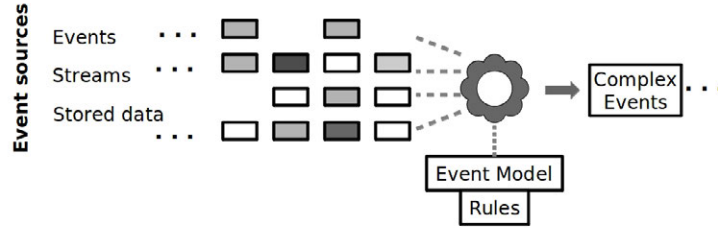


**Figure 2.3:** Common event relationships: precedence, causality and aggregation.

Other relationships such as simultaneity, occurrence in a certain interval, negation, disjunction, etc., can also be used to encode complex events. Event filters are an advanced way for users to be notified only of events matching a given criteria. This criteria can be specified as expressions in a filtering language, including conditions on the type and attributes of an event.

Complex events go beyond filtering and allow defining new types of events based on existing ones, or even creating pattern templates or event patterns, which include

temporal constraints, repetition, ordering conditions, etc. (Cugola and Margara, 2011). We illustrate a complex event generation model in Figure 2.4. Incoming events are streamed and processed according to event models and rules that match the input and produce complex events that are streamed out.



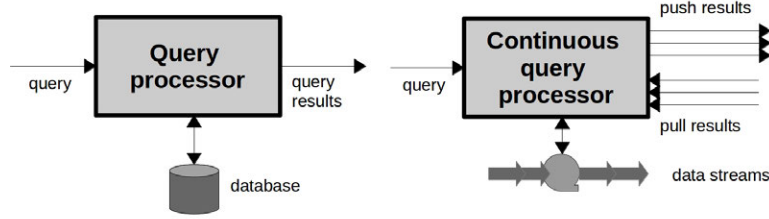
**Figure 2.4:** Complex event generation model.

Although the data model may differ between events and data streams, they also have commonalities. For instance, events normally carry timestamp or interval information and attribute values, and rely on basic data types. In some systems, events can be defined in terms of queries or compositions of filters and patterns over relational streams.

**Continuous processing.** The concept of continuous or standing queries was introduced in (Terry et al., 1992) as “*queries issued once and [...] run continually over the database*”. This idea was developed under the assumption of an append-only database, where traditional querying techniques were not suitable because of inefficiency considering high volumes and velocity of data. The resulting Tapestry system was able to execute monotonic queries periodically, considering only new tuples for each evaluation in order to avoid duplicates but also to guarantee completeness and deterministic behavior.

In subsequent works, this concept changed the execution model in stream-based systems, where data arrival initiates query processing, unlike stored relational databases. In a traditional one-off database query, the result is immediately returned after query evaluation (as in Figure 2.5, left). Conversely a continuous query resides in the query processor, and produces results as soon as streaming data matches the query criteria (Figure 2.5, right). Then the client may actively retrieve the data in pull mode, or the query processor may push it to clients as it becomes available.

Continuous queries, beyond one-time queries or triggers, such as the ones proposed in the Alert system (Schreier et al., 1991), or the trigger-stop queries of OpenCQ (Liu et al., 1999), were further developed in (Golab and Özsu, 2003), including definitions of join semantics and optimizations. Some of the challenges addressed included the pos-



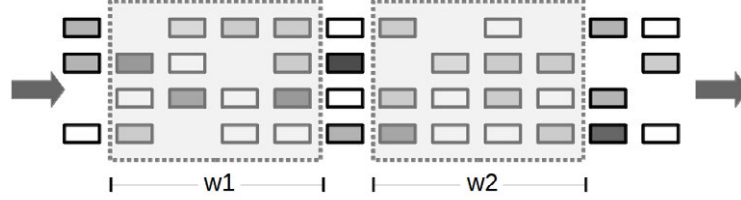
**Figure 2.5:** One-off and continuous query processing models.

sibility of handling thousands of standing queries concurrently, sharing data between query operators and performing static and runtime optimizations. The idea of grouping similar queries, judging by their signature and sharing a global plan within the group, was exploited in NiagaraCQ (Chen et al., 2000), for XML based data streams. TelegraphCQ (Chandrasekaran et al., 2003), built from the previous systems CACQ (Madden et al., 2002) and PSoup (Chandrasekaran and Franklin, 2003), used Eddy operators for routing tuples between query operators. It also promoted sharing through state modules, temporary repositories of homogeneous tuples. Through adaptivity and sharing, TelegraphCQ was able to handle large numbers of simultaneous queries.

Another key problem concerning continuous evaluation is related to the memory capabilities required for query processing. The next textual query example: “*get the current temperature every hour at location X*”, illustrates some common requirements of continuous queries. It is expected to monitor the evolution of a certain value (e.g. temperature), with some given timing constraints (e.g. every hour). In the example the query is expected to provide answers each hour, and past values are not relevant. However, many different queries may be running at the same time, for instance “*get the average temperature of the latest 10 minutes every hour*”, requires the latest 10 minutes of data to compute the results. It has been shown that certain types of queries cannot be answered with bounded memory, and in those cases there are scenarios where the memory requirements are linear with respect to the input streaming data (Arasu et al., 2004a). When memory requirements exceed the available resources, data summarization can also be used to answer a query with approximate results (Gehrke et al., 2001).

**Windows.** The sliding window model is probably the most common and widespread norm for streaming data processing. The idea behind windows is simple: it basically limits the scope of tuples to be considered by the query operators. For instance the window  $w_2$  in Figure 2.6 has a starting point and an offset that limits its scope over certain

stream tuples. A subsequent window  $w1$  will bound data tuples in a latter interval of the stream. Notice that windows may immediately follow each other, overlap, or be non-contiguous in terms of time.



**Figure 2.6:** Windows  $w1$  and  $w2$  over a data stream.

Windows are usually specified in terms of an offset or width, and a slide that indicates the periodicity with which a new window will be created. Considering, for instance, a query that asks for the temperature average in the latest 10 minutes, a window can be set from the current time to the past 10 minutes, and all the remaining data can be excluded. However, for continuous queries, the window *slides* over time, that is, for each execution time, the window boundaries will have moved.

Sliding windows, tumbling windows, landmark windows, update-interval windows and other variations have been present in different systems including Aurora, STREAM, NiagaraCQ, Tribeca or TelegraphCQ. However in many cases these systems lacked clear semantics of the window operators, and the relationships between them. One attempt for an abstract semantics was given in (Arasu et al., 2007), who introduced the concept of *stream-to-relation* operators. These operators take a stream and transform it to a relation with the same schema. This simple conceptualization was successfully used to define different types of windows:

- Time-based windows: Given a time interval  $T$  over a stream  $s$ , the output relation is defined by the window of size  $T$ , sliding over time. The specification of the interval may allow defining punctual windows, tumbling windows, etc.
- Tuple-based windows: Given an integer  $n$  over a stream  $s$ , the output is defined as a relation of size  $n$  of the latest values of  $s$ .
- Partitioned windows: Given an integer  $n$  and a subset  $\vec{a}$  of the attributes of a stream  $s$ , it divides  $s$  in sub-streams, one for each attribute of  $\vec{a}$ . Each sub-stream groups the tuples with the same value on the corresponding attribute of  $\vec{a}$ , and is then limited in number by  $n$ . Finally all windowed sub-streams are merged as final output.

These definitions have been shown to be equivalent or overlap other approaches. The window model has been applied not only for query processing, but also for other purposes such as time-series analysis and approximation of streaming data (Keogh et al., 2004).

### 2.1.2 Continuous Query Languages

Both windows and continuous queries arise from requirements that exceed the capabilities of traditional database languages. While traditional one-off (SQL-like) queries are suitable for stored data, streams require continuous long-lived queries that process and yield results as tuples arrive. To cope with these requirements, several continuous query languages have been designed, for object-oriented, relational-based, and event-based models (Cugola and Margara, 2011). We describe their main features below, and Table 2.1 provides a non-exhaustive comparison of streaming query languages according to their main features. These include the type of data model (e.g. object oriented, data stream or event based), the query type (e.g. declarative, based on relational SQL, etc.), the supported operators (e.g. projections, selections, joins, sequencing, windows, aggregates, etc.) and the ability of combining streaming and stored data.

**Object and XML languages.** Object-oriented languages have also been used for streaming data, adding time-aware methods to objects, so that these can be used in queries. For instance COUGAR (Bonnet et al., 2001) included periodical execution of sensor queries, extending SQL with the `every` keyword, followed by an interval of time. The Tribeca (Sullivan and Heybey, 1998) query language uses a DDL to define objects as types of packets with hierarchies, and these can be used in queries that may specify projections, aggregations or windows. For instance the following window `w1` is defined over a stream `s1`, for intervals of 0.005 on the `ts` attribute.

---

```
stream window w1 on s1
defined by {s1.ts.interval 0.005} is fixed
```

---

Other languages were designed for XML processing, where semi-structured data could be viewed as object definitions. The NiagaraCQ language for XML streams, extended the XML-QL query language with a construct for creating continuous queries and another one for deleting them. The query construction allows indicating the execution interval, as well as start and expiration times. In the web-event streaming data system OpenCQ, the query language extended SQL with trigger and stop conditions, and was also targeted for semi-structured data flows.



Query Languages	Data model	Query Type	Proj/Sel	Conj/disj	Seq ops	Joins	Windows	Aggregates Stored
Tribeca	Object DDL	Imperative rules	Yes	No	No	No	Fixed, Sliding window rules	Yes
COUGAR	Object oriented	Decl In-network	Yes	No	No	No	Fixed	Yes
NiagaraCQ	XML	Decl XML	Yes	No	No	Yes	No	Yes
OpenCQ	Data stream	Decl query + triggers	Yes	Yes	No	Yes	No	Yes
TQL (Tapestry)	Data stream	Decl SQL based	Yes	No	No	No	No	No
Aquery	Data stream	Decl SQL based	Yes	No	No	Yes	Ordered Groups	Yes
StreamQuel	Data stream	Decl SQL based	Yes	No	No	Yes	Yes (loops)	Yes
CQL	Data stream	Decl SQL based	Yes	No	No	Yes	Yes	Yes
TinyDB	Data stream	Decl In-network	Yes	No	No	Yes	No	Yes
SNEEql	Data stream	Decl SQL based/in-network	Yes	No	No	Yes	Yes	Yes
Aurora-Borealis	Data stream	Boxes and arrows/workflow	Yes	No	No	Yes	Yes	Yes
ESL (Stream Mill)	Data stream	Decl SQL based	Yes	No	No	Yes	Yes	Yes
Rapide	Event-based	Pattern matching rules	Sel	Yes	Yes	Yes	No	No
GEM	Event-based	Event declaration and rules	Sel	Yes	Yes	No	Fixed, point in time	No
CEDR	Event-based	Declarative event pattern	Yes	Yes	Yes	Yes	No (AdjustLifetime operator)	Yes
Cayuga	Event-based	Decl SQL based	Yes	Yes	Yes	Yes	No	Yes
Tesla	Event-based	Pattern matching rules	Sel	No	Yes	No	Yes (interval)	Yes
OracleCEP	Event/Data stream	Decl CQL based	Yes	Yes	Yes	Yes	Yes	Yes
SybaseCEP	Event/Data stream	Decl SQL based	Yes	Yes	Yes	Yes	Yes	Yes
StreamBase	Event/Data stream	Decl SQL based	Yes	Yes	Yes	Yes	Yes	Yes
Esper	Event/Data stream	Decl SQL based	Yes	Yes	Yes	Yes	Yes	Yes

Table 2.1: Comparative table of streaming data/event query languages.

**Relational-based languages.** The wide adoption of SQL and relational-based declarative languages in the database industry naturally led most streaming data and continuous query proposals to extend this model.

The Tapestry query language TQL introduced the possibility of “*installing*” queries that would be periodically executed, provided that the query is monotonic (i.e. results do not decrease for new tuples in the input). To allow arbitrary queries, they introduced rewriting algorithms to transform input queries into monotonic ones, for append-only databases. AQuery (Lerner and Shasha, 2003) is one of several query languages that extends relational algebra, in this case, for ordered data. It essentially views relation columns as arrays where order dependent functions (e.g. previous, first, last tuple) can be applied in the query body. Although not exclusively thought for streaming data, AQuery allowed window operators in its query algebra.

StreaQuel is the language used by TelegraphCQ (Chandrasekaran et al., 2003), extending SQL for continuous evaluation. Windows are specified with the `WindowIs` keyword, inside a `for` loop. In the following example, the query requests the average closing price for the MSTF stock symbol, every 5 days, for the last 5 days. The query expires after 50 days and `ST` is the starting time.

---

```
Select AVG(closingPrice)
From ClosingStockPrices
Where stockSymbol = 'MSFT'
for (t = ST; t < ST + 50; t +=5 ){
    WindowIs(ClosingStockPrices, t - 5, t);
}
```

---

The CQL language (Arasu et al., 2006) supports many of the different types of windows and operators of previous languages, and provides well-defined semantics for its evaluation. It has been used in systems such as STREAM, or as a basis for commercial engines as Oracle CEP. The window upper boundary is typically the current time, for instance in CQL the following statement: `S [RANGE T]` denotes the relation consisting of the elements of `S` whose timestamp falls in the latest `T` time units. For example `S [RANGE 30 seconds]` would result in a window consisting of the elements issued in the latest 30 seconds. More complex time windows and special cases (e.g. point-in-time window with the `NOW` keyword, or limitless windows using `UNBOUNDED`) are also considered in CQL. Sliding parameters are another usual feature of time-based windows. The slide specifies how often the window is produced. For instance in CQL the following statement defines a window that slides every `L` time units: `S [RANGE T SLIDE L]`.

Tuple windows are also supported in CQL. For instance `S [Rows 4]` represents a re-

lation with the last 4 elements of the stream  $S$ . A slide parameter can also be specified for tuple-based windows. Apart from performing stream-to-relation transformations it is often required to transform relations to streams. The main operators available for this purpose are RSTREAM, ISTREAM and DSTREAM. The ISTREAM operator applied to a relation  $R$  at time  $t$ , produces a stream that contains all tuples of  $R$  that are in  $R$  at time  $t$  but not in  $t - 1$ . This behavior informally specifies that the produced stream contains only the inserted elements in a relation at each time. The DSTREAM operator applied to a relation  $R$  at time  $t$  produces a stream that contains all tuples of  $R$  that are not in  $R$  at time  $t$  but were there in time  $t - 1$ . Therefore the resulting stream contains the deleted elements of  $R$  at each time. Finally the RSTREAM operator applied to a relation  $R$  produces a stream that contains all tuples of  $R$ .

**Sensor Network Query Languages.** Sensor networks, with reduced storage and processing capabilities, are typically configured for data retrieval using programming languages, such as nesC (Gay et al., 2003) or other subsets of C. However, there is a trend for using declarative languages to specify data requirements. While these languages may be similar to the relational based ones described below, they exhibit some distinct features.

COUGAR (Bonnet et al., 2001) was devised specifically as a sensor-based system, and pioneered in the definition of an architecture that enables the use of declarative queries for sensor query processing. Nevertheless, the language specification and semantics are not described in the available publications. Another example of such query language is the one of TinyDB (Madden et al., 2005), a system that implements acquisitional query processing (i.e. data is acquired only if needed by the queries). The language extends SQL with sampling constructs, but omits time or tuple windows. A more recent approach is SNEEql (Galpin et al., 2011), which has a well defined, unified semantics for declarative expressions of data needs over event-streams, acquisitional-streams, and stored data. Apart from the window constructs, it allows specifying QoS expectations (e.g. acquisition rate, delivery time), which may affect the query optimization process and translation of the query to sensor network native code.

**Complex event processing languages.** Complex event processors add some operators to the filters, grouping, aggregators and windows that we have seen previously. These additional operators are needed to allow defining queries that exploit the causality, temporal simultaneousness (conjunctive or disjunctive), and compositional relationships of events, that we saw in Section 2.1.1. For instance in the Rapide (Luckham and

Vera, 1995) language the pattern  $A(?i)$  and  $B(?i)$  matches events  $A$  and  $B$  having the same attribute. Also, temporal relationships can be expressed between events, for example  $A < B$  denotes an event  $A$  preceding  $B$ . Causality between events can also be included in patterns, using the  $A \rightarrow B$  notation.

Composite events can be constructed in languages such as GEM (Mansouri-Samani and Sloman, 1997) using sequence operators:  $A ; B$ , meaning that event  $A$  occurs before  $B$ , or conjunction and disjunction of events ( $A \& B$  and  $A | B$  respectively). Event conjunction represents the occurrence of both events in the same interval, while disjunction in these languages is the occurrence of either of them. Negation is another construct that has been defined in event-based languages such as CEDR (Barga et al., 2007). For example, the expression  $\text{UNLESS}(E1, E2, t)$  matches the occurrence of event  $E1$  if it is followed by no  $E2$  event in the next  $t$  time units. These operators can be combined in event processing languages, for instance applying a negation to an event sequence, or defining a conditional sequence pattern. For instance, in Cayuga (Brenna et al., 2007) the expression  $E1 \text{ NEXT } \{\theta\} E2$  matches an event  $E1$  followed by  $E2$  if the predicate  $\theta$  is satisfied.

**Procedural queries.** In a different approach, Aurora-Borealis (Abadi et al., 2003) relies on streaming data processing based on workflow operations. These are specified using arrows and boxes created by the user, making it a procedural language, rather than the declarative ones previously described.

### 2.1.3 System Implementations

In this section we provide a brief overview of several systems implementing streaming data and event query approaches. We classified the systems into four groups, corresponding to data stream management systems, complex event processors, sensor network query processing systems and sensor middleware.

**Data Stream Management Systems.** Several Data Stream Management Systems (DSMS) or systems managing continuous data, have been designed and built in the past years. Most of them exploit the power of continuous query languages like those described in the previous section, and they generally provide two types of data access mechanisms for streams: pull and push-based. In pull-based access, the client periodically retrieves the data from the DSMS, while in push-based access, it subscribes to a data resource (typically the continuous results of a query) and receives notifications as data become

available. Some of the key features of most DSMS include execution of multiple concurrent continuous queries, online processing of incoming data streams, data summarization, declarative query execution, query plan adaptivity, etc.

As we saw previously, Tapestry (Terry et al., 1992) introduced continuous query semantics as incremental database queries, i.e. for each query execution, only the tuples produced after the last execution are considered. In this way, the continuous query results could be seen as the merge over time of all the individual query answers. The language used by Tapestry was essentially a dialect of SQL, whose queries were limited to a single stream, and did not provide windowing or other streaming operators. This idea has been expanded to architectures where new streams can be generated after applying operators such as windows, filters and aggregators to an initial stream of data, as in Tribeca (Sullivan and Heybey, 1998) and most of the subsequent DSMS.

However, streaming data applications also require querying static or stored data, and one of the challenges in this line is centered on the definition of coherent semantics that allow this type of mixed data management. Using materialized views over streams is one of the early approaches for combining stored and streaming data, as in the Chronicle model (Jagadish et al., 1995) algebra. However, it lacks window operators other than the *latest* tuple. Later approaches, namely STREAM (Arasu et al., 2007), define the semantics of query evaluation as extensions to relational algebra operators. To solve the problem of blocking operators in the presence of infinite streams, STREAM uses stream-to-relation operators that transform unbounded streams of tuples into finite sets that can be evaluated by standard operators. Windows are one example of such operators, either time-based or count-based.

Windows are not directly supported in early continuous query systems, although the behaviour of simple windows can be simulated with other mechanisms. For example, in OpenCQ (Liu et al., 1999) it is possible to define trigger-stop conditions on a standing query, so that the query throws results only if the condition is met. These conditions may include expressions comparing a value and a threshold (e.g. `qty_on_hand + qty_on_order < threshold`) or periodic executions (e.g. `6:00 p.m. everyday`). In NiagaraCQ (Chen et al., 2000), queries may include timer-based execution parameters, meaning that the query is fired with time-based conditions (e.g. every certain time interval, or a specific punctual time). Another early alternative to windows can be found in Gigascope (Cranor et al., 2003). Its query language GSQL relies on restrictions on blocking operators, so that these must be constrained by non-decreasing ordered attributes.

Beyond language expressiveness, DSMS's research has worked heavily on scalability, performance and query optimization. Hosting a large number of continuously running queries efficiently requires several optimization techniques. For instance, NiagaraCQ may scale to thousands of concurrent queries, thanks to incremental group optimizations based on query signatures. Aurora (Abadi et al., 2003), a workflow-based query DSMS, and its successor Borealis (Abadi et al., 2005), includes monitor components that compute statistics and evaluate QoS parameters. With this information it can apply different types of optimizers at the local, neighborhood and global level, considering a distributed streaming system environment.

TelegraphCQ (Chandrasekaran et al., 2003) adopts a heavily adaptive optimization approach that includes Eddy routing operators that choose where to direct every input tuple, based on a policy. The system allows querying over streams and tables, and relies on windows for applying blocking operators such as joins, so that the unbounded nature of the streams can be reduced in continuous evaluation.

The Stanford STREAM (Arasu et al., 2007) system also implements adaptivity at various levels, relying on a Profiler module that computes statistics and a Reoptimizer that may change the query plans according to the current statistics and constraints.

Other optimization techniques are specific for some operators, such as user-defined window aggregates in Stream Mill (Bai et al., 2006). Window aggregate queries can be rewritten so that they can include explicit expiration policies that significantly reduce the data to be processed. In the case of CAPE (Continuous Adaptive Query Processing Engine) (Rundensteiner et al., 2004), it focuses on adaptivity in continuous query processing at different levels. It uses adaptive operators such as PJoin that are memory efficient for join probing. It also relies on adaptive operator scheduling algorithms that take into account QoS requirements at runtime. CAPE also considers plan migration strategies at runtime, which are chosen depending on several parameters including selectivities, windows and data rates.

Finally, there have also been efforts for managing large massively parallel data streams. StreamCloud (Gulisano et al., 2010) is a stream processing system for shared-nothing clusters, which provides automatic parallelization of queries through a query compiler. StreamCloud relies on an underlying DSMS (e.g. Borealis) and rewrites the queries as parallel sub-queries, with load-balancers and input-mergers implemented as conventional streaming operators (filter and union respectively).

**Complex-Event Processors.** These systems (CEP) emerged at the same time than DSMS, and in many cases they overlap in terms of requirements and functionality. While

many DSMS are also shipped as CEP, and the division is somewhat blurry, the focus in the latter is on capturing events with more complex structures. On the other hand streams are closer in structure to relational schemas and therefore the types of queries and operators needed may differ. CEP also emphasizes on pattern matching in query processing. This may include event order patterns, repetition of events over time, finding sequences of events, etc. (Chandy et al., 2011).

As we saw in Section 2.1.2, CEP query languages include pattern expressions in queries including precedence, conjunction and disjunction in a time interval and causality, among others. Rapide (Luckham and Vera, 1995) is an early example of such systems, and although meant primarily for event simulation, it presents some of the main features of event processing in its definition language. Event processing languages such as GEM (Mansouri-Samani and Sloman, 1997), allow defining complex events from basic ones through a series of operators. Basic events in GEM have a type, a timestamp, a source (originator) and a set of data values. For instance, the following event of type *warning* was generated by *sensor1* at the indicated time and contains an attribute with a text label and a value of  $-4.5$ .

---

```
warning "sensor1" [10:30 27/6/95]
  (-4.5, "Temperature too low")
```

---

Time-based operators are commonly used for defining complex events. In GEM, for instance, the next event expression represents an event occurring when an event *A* has its timestamp in a given time interval.

---

```
A when ([10:00] < @A ) && ( @A < [12:00])
```

---

Complex events can be defined using other operators such as conjunction, disjunction, precedence, causality, etc. These are present in more recent CEP, such as CEDR (Barga et al., 2007), Cayuga (Brenna et al., 2007) or Tesla (Cugola and Margara, 2010). Most of these CEP, including Cayuga, Tesla, DistCED, Sase and NextCEP, use non deterministic automata for pattern detection and for representing the language semantics. For example in Tesla, event definition rules are translated into automata models, and instances of these models are produced during evaluation. Transitions that do not satisfy the rule constraints are discarded, and only the automata instances that reach their final state will match an existing event (Cugola and Margara, 2010).

Some of the previously mentioned systems: Aurora-Borealis, STREAM or TelegraphCQ, have also been cataloged as CEP. Other, mostly commercial systems, are usually sold as

CEP and include Oracle CEP<sup>1</sup>, Sybase CEP<sup>2</sup>, StreamBase<sup>3</sup> and Esper<sup>4</sup>.

In the remainder of this thesis we will not insist on the differences between DSMS and CEP, and use them as different technologies that provide query processing capabilities for streaming data tuples or events. We will just make the distinction when there are technical or particular features to consider or highlight.

**Sensor Network Query Processing Systems.** Sensor Networks are one of the main sources of streaming data (Abadi et al., 2004). Sensor networks consist of multiple interconnected nodes that are able to sense and capture different kinds of data. It is evidently necessary to provide means to query data collected by these networks.

While in DSMS and CEP data management and querying are usually centralized in a query or event processor, sensor in-network processing distributes the query computation across different nodes. In the first case data is supposed to arrive to the system in a known homogenized form while in the second, all nodes are configured to handle the same type of data (e.g. in TinyDB, a single “sensors” table (Madden et al., 2005)).

In order to address these requirements, research has produced sensor network query processing engines such as TinyDB<sup>5</sup>, COUGAR (Yao and Gehrke, 2002) and SNEE (Galpin et al., 2009). These processors use declarative query languages for continuous data like the ones we described earlier in this section. Declarative queries describe logically the set of data that is to be collected, and leave to the engine to determine the algorithms and plans that are needed to get the data from the different nodes of the sensor network. These engines normally apply optimization techniques in order to efficiently gather the data values from sensor nodes.

Sensor networks are characterized by strong resource limitations such as energy consumption, computing power and storage capabilities. Architectures for query optimization in these constrained scenarios have surfaced (Galpin et al., 2008; Madden et al., 2005), showing that even with such limitations it is still possible to use rich and expressive declarative query languages.

For example, the SNEE system uses queries expressed in the SNEEQl language that are optimized for evaluation within a sensor network over acquisitional-streams by the SNEE compiler. SNEE also enables query evaluation over event-streams either within the sensor network (in-network query processing) or on computational hardware outside

---

<sup>1</sup>Oracle CEP: <http://www.oracle.com/technetwork/middleware/complex-event-processing/overview>

<sup>2</sup>Sybase CEP: <http://www.sybase.com/products/financialservicesolutions/complex-event-processing>

<sup>3</sup>StreamBase: <http://www.streambase.com/>

<sup>4</sup>Esper and Event Processing Language EPL: <http://esper.codehaus.org/>

<sup>5</sup>TinyDB Project: <http://telegraph.cs.berkeley.edu/tinydb/>



the sensor network.

**Sensor Data Middleware.** Sensor data middleware is the software that mediates the interactions between sensor networks and applications. Sensor query processing engines as the ones previously mentioned are suited for managing queries at the local level, and usually communicate through a gateway to a centralized or distributed middleware. In this way, applications do not need to deal with the internals of network topologies or other details, and focus on data-centric operations.

Centralized DSMS such as STREAM or TelegraphCQ may be used as the core of such middleware, but may not scale out, having a single point of execution. Systems such as Borealis circumvent this problem by introducing distributed query processing at the core of its query engine (Abadi et al., 2005). However, this approach still requires previous agreement on the data schemas.

Other proposals use Peer-to-Peer, hierarchical or other types of distributed data management, using heterogeneous schemas and formats, but relying on adapters, proxies or wrappers for integration (Doan and Halevy, 2005; Gurgun et al., 2008). In these systems, each node is responsible for managing and processing some of the sensor streams, usually due to geographical proximity. The distribution criteria may also obey load balancing principles, or a hierarchical structure. Some of the queries may require processing only in one of the nodes (local processing), for instance, when asking for the latest temperature in a certain location. However, when it comes to integrating different sensor streams, these systems interact through gateways or service interfaces.

The Open Geospatial Consortium Sensor Web Enablement (OGC-SWE) (Botts et al., 2006) defines a set of XML data models for describing sensors and their data (see Section 2.4), as well as a set of web services for publishing sensor data, locating data, and receiving alerts. The SOS<sup>1</sup> service main operations are listed below:

- *GetCapabilities*: provides metadata about the SOS service, including operations available, filter capabilities (e.g. spatial, temporal, etc.) and observation offerings (e.g. observation type, feature of interest, response format, etc.)
- *DescribeSensor*: provides metadata about the sensors
- *GetObservation*: provides data access to observations. The request may specify filtering operations over the data (spatial, temporal, etc.).

---

<sup>1</sup>Sensor Observation Service: <http://www.opengeospatial.org/standards/sos>

The OGC standards serve as a model that implementers can use as a base for integration and fusion of sensor data. A reference implementation of the framework was developed in the SANY project (Schimak and Havlik, 2009). The framework can be seen to provide support for relevant data source discovery and display information from different and diverse sensor data sources. However, data access patterns are limited by the service interfaces and there is no support for declarative query languages. Data is published according to their XML data models, which is not always possible with autonomous data sources and there is support for non-mediated merging of sensor and stored data but not for correlating it. Henson et al. (2009) have extended the sensor observation service by semantically annotating the data.

Standardization of operations and specially of metadata and observation data is a major improvement over earlier approaches based on XML-based schemas. For example the agent-based IrisNet (Gibbons et al., 2003) system was based on XML for representing sensor data and contextual metadata, and XPATH as a query language.

Other service oriented middleware, such as SStreamWare (Gurgen et al., 2008), SenSer (Paulino and Santos, 2011) or Hourglass (Shneidman et al., 2004), expose mostly filtering capabilities for retrieving observations. In Hourglass, every node may implement registry, orchestration (called *circuits*) and data producer interfaces. Through these circuits, Hourglass allows defining workflows with XML files, but does not provide a rich streaming query language as the ones we studied in section 2.1.2. SenSer provides a logical layer that allows collecting data from a sink, pipelining filter operators and archiving data in a repository. A similar approach is followed in the OSGI-based<sup>1</sup> SSstreaMWare, which proposes a hierarchical structure of query processing nodes, according to their geographical location. Queries, expressed in a SQL-like language, are evaluated by the different nodes, communicating via web services.

SenseWrap (Evensen and Meling, 2009) uses the Zero configuration networking protocols (Zeroconf) to provide a lightweight service-oriented middleware for heterogeneous sensors. This architecture focuses on sensor discovery through implementation of device-specific *DiscoverSensors* components. Clients view the sensors through virtual services that they can interact with using different protocol adapters (e.g. UDP HTTP, SOAP, etc.).

Global Sensor Networks (GSN) (Aberer et al., 2006) provides a peer-to-peer sensor middleware that hides sensor deployment specificities behind the concept of *virtual sensors*. Sensor proprietary formats, means of communication and data structures are hidden behind the virtual sensor abstractions, and require simple wrappers and configu-

---

<sup>1</sup>OSGi Alliance: <http://www.osgi.org>

ration to be deployed in GSN. GSN peers can communicate with each other through streaming web interfaces, creating virtual sensors that reuse data from other GSN nodes. A Web Service interface and a REST API are also provided to pose queries to the sensor data streams. An example of a URL query that requests wind speed values for the latest 10 minutes is given in Listing 2.1.3 (assuming the current time is 15/09/2011-15:00:00).

---

```
http://montblanc.slf.ch:22001/multidata?vs[0]=wind_sensor
    &field[0]=wind_speed
    &from=15/09/2011+05:00:00
    &to=15/09/2011+15:00:00
```

---

In this example the GSN server (montblanc.slf.ch:22001) exposes a virtual sensor named `wind_sensor` whose values are fed to the GSN system through some data adapter. This URL query is equivalent to the previous ones except that the time window has to be set explicitly. It is also possible to define a query as continuous and set sliding windows, but only through configuration. More complex queries can be built using these URLs, including selection constraints or joins with other sensors and stored tables.

As a summary, we provide in Table 2.2, a comparison of some of the main DSMS, CEP and sensor middleware discussed previously. We emphasize the salient features of each one, although it is not an exhaustive list. Note that there are solutions for dealing with generic streaming data, most notably DSMS, and that for specific event-based and query-pattern requirements, CEPs are well suited. However most recent commercial products blend these two into comprehensive solutions, usually integrated with existing stored database systems. In some cases, particularly for sensor networks, in-network query processing is suitable for providing high-level abstractions for controlling the data acquisition processes (e.g. rates, delay, etc.).

On the other end, access to clients is commonly provided through middleware service-oriented solutions, where concerns of standardization, discovery and distribution are some of the main challenges. In brief, access to streaming and sensor data is still hidden by layers of heterogeneous services, data models, schemas and query languages for users and applications.

## 2.2 Ontology-based Access to Relational Data Sources

The goal of ontology-based data access is to generate semantic content from existing relational data sources (Sahoo et al., 2009). There is a considerably large amount of work in

## 2.2. Ontology-based Access to Relational Data Sources

Systems	Type	Data streams	Query Evaluation	Optimization	Implementation
Tapestry	Append-only DB	Relational streams	Continuous query (monotone query rewriting over DBMS)	Only tuples latest execution	Translate to date-triggered stored procedures on Sybase
Tribeca	Network traffic	Single (network) stream	Compilation of query to executable graph	Combination of operators, overlapping windows	Live or taped network analysis dataflow software
Chronicle	Data model & algebra	Relations, views, sequences	Periodic persistent views	-	Chronicle algebra
OpenCQ	DSMS	Wrapped stream events	Trigger-stop conditions	Multiple condition, incremental condition evaluation	Continual query, trigger condition evaluation, time-based event detector.
NiagaraCQ	DSMS	XML	Time-triggered periodic queries	Incremental group optimization	XML source monitoring, event detector
Borealis	DSMS	Boxes and arrows	Continuous query processing, dynamic query modification	Local, neighborhood, global	Boxes and arrows workflow-based
Gigascopie	Network traffic	Relational streams	Code generation from GSQL queries	NIC code-level specific optimizations	C code generation
TelegraphCQ	DSMS	Relational streams	Continuous query processing	Eddy adaptive operators	Cont. query processing extending PostgreSQL
STREAM	DSMS	Relational streams	Continuous query processing	Continuous query processing	Continuous query processing exploiting stream-to-relational operators
Stream Mill	DSMS	Relational streams	Continuous query processing	Rewrite window aggregates	Translation to C functions
CAPE	DSMS	Relational streams	Continuous query processing	Plan, Adaptive operation scheduling, Plan reorganizer	Adaptive query processing
TinyDB	In-network query processing	Single stream	Query partition and dissemination over network	Power-based query optimization	Acquisitional query processor for sensor networks
Cougar	In-network query processing	In-memory tables	Query plan dissemination to sensor nodes	Multi-query optimization	Query proxy layer on sensor nodes
SNEE	In-network query processing	Relational streams	Query plan dissemination to sensor nodes	Static optimization, QoS based optimization	Translation to native nesC code
Rapide	EADL architecture simulation	Events, posets	Simulation, event pattern matching	Compilation optimization.	Compile architecture model, run-time library produces an executable simulation
GEM	CEP	Events, composite events	Rule maintains event history for specified window. Explicit termination for scheduled time events.	Delay management.	Interpreted rules, generate an Event evaluation tree, incremental evaluation.
CEDR	CEP	Events	Compile queries to runtime operator algebra	Logical query optimization	Only semantics of runtime operators provided
Cayuga	CEP	Relational streams (fixed schema)	Epoch based processing	Not described	Finite state automata
Tesla	CEP	Events	Incremental event detection based on automata	Not described	Automaton model matching against standing rules
OracleCEP	DSMS/CEP	Relational streams/events	Continuous queries evaluated by event processor instances (dataflow configuration)	Static optimization, indices, row and window sharing.	Java-based runtime, multiple OracleCEP applications
SybaseCEP	DSMS/CEP	Relational streams	Continuous query processing, from input source adapters	Static optimization, indices, row and window sharing.	Multithreaded C++ server, adapter-based input sources
StreamBase	DSMS/CEP	Relational streams	Compile continuous queries at runtime to execution units	Static optimization, indices, row and window sharing.	Multithreaded server
Esper	DSMS/CEP	Events	Continuous query processing, state machine representation	Static optimization, indices, row and window sharing.	Java, .NET event processing runtime
OGC-SWE	Middleware	OGC messages	Service specification only.	N/A	N/A
SstreamWare	Middleware	Internal schema	Proxy sub-query evaluation	N/A	OSGi Declarative services
Cosm	Middleware	JSON data items	Service requests, internal execution	N/A	REST
Hourglass	Middleware	XML	Application-specific code, plugged to a service layer, form data producer proxies	N/A	TCP connections between Hourglass Java-based nodes
GSN	Middleware	XML/Json/CSV	Service requests/distributed. Relational in-memory processing	N/A	REST/SOAP

**Table 2.2:** Comparative tables of DSMS, CEP and streaming sensor middleware.

this area, as most stored data in the web is currently preserved in relational databases, and as some data integration approaches are based on the existence of ontologies.

In this section, we discuss the main challenges and approaches for ontology-based data access, including data generation and querying, establishment of mappings, and different implementations.

### 2.2.1 Data Generation and Querying

Ontologies can be specified in different languages and standards including OWL or RDFS, depending on the expressiveness needed or the required complexity of reasoning tasks. Regardless of these considerations, RDF<sup>1</sup> is widely used as a general modelling standard for encoding semantic data, in its diverse serialization formats (including RDF-XML, TURTLE, etc.). Hence, systems producing RDF data from relational databases are often labeled as RDB2RDF tools or frameworks.

Research on the subject has produced many approaches to building such systems. We highlight two main alternatives: the *virtualization* and *materialization* approaches (Ibrahim et al., 2005). In the **virtualization** approach queries are posed over an ontological schema and a mediator component identifies the sources that will be needed to produce the answer. Then a series of appropriate sub-queries are automatically created for these sources (typically in SQL) and evaluated. The results of each of these queries are retrieved, post-processed and transformed to the mediated schema and returned to the caller application. The objective of these systems is to allow users to construct queries over an ontology (e.g. in SPARQL, see Section 4.3.1), hiding the data structure and internal details of the underlying data source.

This alternative allows answering queries on demand, even if the update rate of data is high and the queries are potentially arbitrary. As the queries are rewritten dynamically for each of the sources, it is possible to retrieve any portion of data exposed through the mediated schema. However, the transformations on the queries and the processing of the data from the sources to the mediated schema may be complex and incur in performance issues. Another potential problem is data source availability, as the queries are performed live; if the source is unreachable then the queries may not be able to be completed. Even if the source is available, latency caused by networking or connection problems may increase the query response time to unacceptable situations.

---

<sup>1</sup>Resource Description Framework: <http://www.w3.org/RDF/>

The **materialization** approach differs from the virtualized one in that it extracts and transforms the data from the sources in bulk-load mode, instead of performing on-the-fly conversions each time a query is being executed. Instead, relevant data is first identified, and extracted from the multiple data sources in a batch process operation, similar to an ETL process (Extract-Transform-Load). The extracted and transformed data is then stored in a data store or warehouse where it can be accessed by the external applications. The data can be updated periodically depending on the requirements. Under this approach, the time consumed in retrieving the data from the sources is moved to the off-line phase of updating the centralized data warehouse. Therefore when the external applications query the system, results can be accessed almost immediately. However the data may be stale in occasions, and it may not be possible to access all possible pieces of data, as the warehouse only stores selected materialized views of the original sources.

In both approaches, a mediator component is introduced as a major feature of the integration architecture. For virtualization, this mediator is in charge of transforming the original query into sub-queries for the sources and then transforming the results back to the mediated schema. In materialization approaches, the mediator is in charge of data transformation. These processes are performed thanks to mapping definitions that establish relationships between the mediated and source schemas.

### 2.2.2 Mappings for RDB2RDF

Regardless of the choice of virtualization or materialization approaches, these systems require rules (or mappings) that relate the original and target data models. These *mappings* describe the equivalences between terms in the relational schema and terms in the ontology. They are normally encoded as rule expressions, or as part of mapping languages especially devised for this purpose, although in some cases they are use-case specific or directly hard-coded in a wrapper or adapter. We will focus on the first case, considering the need for declarative mapping definitions that can be reused, shared and interpreted by different engines and applications.

From an abstract point of view, there are three main alternatives for defining these mappings (Ibrahim et al., 2005; Lenzerini, 2002): Global-as-view (GAV), Local-as-view (LAV) and a combination of both, GLAV. In the LAV approach, each of the source schemas is represented as a view in terms of the global schema. Consider a system  $I$  represented

as:

$$I = \langle G, S, M \rangle$$

Where  $G$  is the global schema,  $S$  is a source schema and  $M$  is the mapping between  $G$  and  $S$ , then the mapping assertions in  $M$  will be a set of elements of the form:

$$s \rightarrow q_G$$

Where  $s$  is an element of  $S$  and  $q_G$  is a query over the global schema  $G$ . This approach is useful if the global schema is well established and if the sources suffer modifications constantly. Notice that changes in the sources do not affect the global schema. However, the query processing is not obvious, as it is not explicitly stated how to obtain the data in the mapping definition. Query rewriting techniques such as query answering using views can be used in this approach (Halevy, 2001). Examples of LAV based works are DWQ (Calvanese et al., 1998), InfoMaster (Genesereth et al., 1997), Information Manifold (Levy, 1998) and PICSEL (Goasdoué et al., 2000), among others.

The other approach, GAV, defines the mappings in the opposite way. The global schema elements are represented as views over the source schemas. The mapping definition explicitly defines how to query the sources to obtain the desired information in terms of the global schema. Following the system representation  $I = \langle G, S, M \rangle$ , in the GAV approach the mapping assertions are elements of the form:

$$g \rightarrow q_S$$

Where  $g$  is an element of the global schema  $G$  and is expressed as a view  $q_S$ , a query on the source schema. The advantage is that the mapping itself already indicates how to query the sources to obtain the data, so the processor can directly use this information to perform the query rewriting. The main disadvantage is that in case of changes or additions on the sources -e.g. a new source is added- then the global schema may suffer changes and this may affect other mapping definitions. Examples of GAV based works are SIMS (Arens et al., 1993), TSIMMIS (Garcia-Molina et al., 1997), Carnot (Huhns et al., 1993), Gestalt (Ramakrishnan and Silberschatz, 1998), MOMIS (Beneventano et al., 2007), IBIS (Calì et al., 2003a), or DIS@DIS (Calì et al., 2003b).

For the sake of completeness we can mention the GLAV (Global-Local-as-view) ap-

proach, which is a generalization of the previous two. This formalism allows more expressive mappings than LAV and GAV combined, and it has been shown to reach the limits of tractability of these description languages (Friedman et al., 1999).

**R2RML Mapping Language.** Concrete implementations of mappings have been proposed in the community, enabling the description of RDF triples from relational databases. Even though some of these efforts met a certain degree of usage, they all had their own format and features. The need for a common standard prompted the W3C RDB2RDF Working Group<sup>1</sup> to define the R2RML<sup>2</sup> language for describing mappings from a relational database to RDF graphs. R2RML allows creating custom mappings that take one, several or all tables from a database, or even logical tables represented as SQL queries as an input. From this logical table, a triple mapping (TriplesMap) in R2RML generates triples with a subject defined by a SubjectMap. This SubjectMap defines how the URI of the subject is generated (e.g. a constant value, with a template applied to a table column, etc.). Then the predicate and object of the triples are defined by one or more PredicateObjectMap elements.

For example, the following R2RML TriplesMap defines how triples can be produced from a sensors relational table<sup>3</sup>. The table name is specified using the `rr:logicalTable` property, although we could use a view with a SQL statement instead. This mapping has a SubjectMap (identified as `:sensorSubject`) and two PredicateObjectMaps (`:identifierPO` and `:locationPO`).

---

```
:sensorsMap a rr:TriplesMap;
  rr:logicalTable      [rr:tableName "sensors"];
  rr:subjectMap        :sensorSubject;
  rr:predicateObjectMap :identifierPO;
  rr:predicateObjectMap :locationPO.

:sensorSubject a rr:SubjectMap;
  rr:template "http://sensorgrid4env.eu/ns#data/sensor/id/{sensorid}";
  rr:class ssn:Sensor.
```

---

Therefore for each tuple of sensors, all corresponding triples will be created with a subject defined by the SubjectMap `rr:template`. It will form a URI by substituting the `sensorid` column value into the string template. A special triple will be generated with the `rdf:type` property and the fixed `ssn:Sensor` as object. In addition, for each tuple, two more triples will be generated, one per PredicateObjectMap. These definitions are given

---

<sup>1</sup><http://www.w3.org/2001/sw/rdb2rdf/>

<sup>2</sup>W3C Recommendation R2RML: [www.w3.org/TR/r2rml](http://www.w3.org/TR/r2rml)

<sup>3</sup>We use RDF abbreviated namespaces for brevity.



below. For example, for the first PredicateObjectMap, the triple predicate is fixed as `dc:identifier` and the object will be the value of the `sensorname` column of the `sensors` table.

---

```
:identifierPO a rr:PredicateObjectMap;
  rr:predicateMap [ rr:predicate dc:identifier ];
  rr:objectMap    [ rr:column "sensorname" ].

:locationPO a rr:PredicateObjectMap;
  rr:predicateMap [ rr:predicate dul:hasLocation ];
  rr:objectMap    [ rr:object dbpedia:Davos ].
```

---

### 2.2.3 RDB2RDF Implementations

Several systems exist to provide ontology-based access to stored data, mainly in the form of relational databases, as described in [Sahoo et al. \(2009\)](#). A simple approach is to first generate a syntactical translation of the database schema to an ontological representation. Although the resulting ontology has no real semantics, it may be argued that this is a first step to create an intermediate ontology model and could later be mapped to a real domain ontology ([Lubyte and Tessaris, 2009](#)). SquirrelRDF ([Seaborne et al., 2007](#)) takes this approach through a direct mapping of the relational database schema. This mapping is generated using the table names, columns and keys so that when the RDF is produced, SPARQL queries can be executed against the new repository. Other approaches use not only the schema but also the data to generate a target ontology, as in RDBToOnto ([Cerbah, 2008](#)), although it lacks a query mechanism to the database data. Relational.OWL ([Pérez de Laborda and Conrad, 2006](#)) also builds a direct-mapped ontology from the database schema, but in a second step maps it to a mediated ontology using SPARQL.

Direct mapping proposals such as the one proposed by the W3C RDB2RDF working group<sup>1</sup> are examples of this approach, which yields automatic and immediate transformation to RDF. This type of mappings has been further studied in ([Sequeda et al., 2012](#)), in particular the information and query preservation properties of mappings. Direct mappings provide a default transformation mechanism from a relational database to RDF, without needing to explicitly define mapping rules. The query preservation property of the proposed direct mapping is related to the ability to translate every query over the database to a query over the mapping result. In addition, the information preservation property guarantees that the database can be reconstructed from the mapping

---

<sup>1</sup>W3C RDB2RDF Direct Mapping: <http://www.w3.org/2001/sw/rdb2rdf/directMapping/>

result.

Although automatic generation of ontologies and mappings can be useful in simple scenarios, for complex ones it is a limited approach. User expert knowledge may become necessary for complex mapping definitions, but it is also necessary to provide well defined languages that express those mappings.

Some approaches rely on configuration files that serve as GAV mappings, where the relational data is extracted using SQL query expressions. The SPASQL implementation (Prud'hommeaux, 2007) is one example of such systems, in this case limited to the MySQL DBMS. Triplify (Auer et al., 2009) also uses SQL views but relies on certain conventions to generate RDF data: e.g. a column is intended to represent an instance URI, column names of the views are used to generate the predicate URIs, view cells must contain data values or references to entities in a view, etc. In Virtuoso (Erling and Mikhailov, 2007), the mappings are given by a meta-schema based on Quad Map Patterns that defines transformations from relational columns into triples.

Other RDB2RDF tools include the D2R platform (Bizer and Cyganiak, 2007), whose mapping language is D2RQ. Similar features are provided by the commercial tool Spyder<sup>1</sup>, which adds parallel processing, high speed results transfer and wider support for database platforms and formats. The R<sub>2</sub>O language (Barrasa et al., 2004), used by the ODEMapster system differs from most other systems in that it does not use SQL for defining views, but provides its own set of declarative constructs independently of DBMS-specific dialects. This system has evolved into Morph, one of the early implementors of R2RML and its set of test cases<sup>2</sup>. Some of these systems have been lately adapted to support R2RML, given its W3C recommendation status. In fact, many of these previous efforts were used by the W3C RDB2RDF group to design R2RML.

Finally, besides simple RDB2RDF generation and querying, there are also some approaches that on reasoning for ontology-based data access. MASTRO (Poggi et al., 2008) is a description logic reasoner for ontologies whose data is accessed through mappings in an external source. The reasoner works over the *DL-Lite<sub>ℳ</sub>* language, a fragment of OWL-DL, and the mappings are specified through assertions that include SQL queries over the database. Query rewriting techniques have also been studied to exploit SPARQL queries, having a database for storage. These exploit an ontology TBox to rewrite an initial query as union of conjunctive queries, using saturation-based algorithms (Calvanese et al., 2005). Optimizations aiming at reducing the number of rewritten queries

---

<sup>1</sup>Revelytix Spyder: <http://www.revelytix.com/content/spyder>

<sup>2</sup>RDB2RDF Implementation report: <http://www.w3.org/2001/sw/rd2rdf/implementation-report/>

have also been produced (Pérez-Urbina et al., 2009), but these fall out of the scope of our work. Along this line, Presto (Rosati and Almatelli, 2010) addresses the problem of query rewriting on *DL-Lite<sub>R</sub>*, i.e. not including expressions of the form  $\exists R.B$ , what makes the search for *most-general subsumees* computationally tractable. Prexto (Rosati, 2012) adds extensional constraints into consideration, reducing the size of the rewritten query using concept and role disjointness assertions as well as role functionality assertions.

None of these approaches has applied ontology-based access to *data streams* using RDB2RDF mappings.

## 2.3 Extensions for RDF Stream Querying

The streaming data query languages and systems proposed and implemented in the last years are to a large extent based on relational models and languages. Therefore they are concerned with the same problems in terms of heterogeneity and lack of explicit semantics that enable advanced querying and reasoning.

In this section we discuss the main proposals for extending SPARQL-the W3C recommendation for querying RDF- to query data streams, and also some of the principles for publishing sensor data as Linked Data, which is a widespread method for RDF publishing on the Web.

### 2.3.1 SPARQL Extensions for Streams

Streaming data extensions for SPARQL include operators such as windows, the ability to deal with RDF streams, and the possibility of declaring continuous queries.

Early approaches such as  $\tau$ SPARQL (Tappolet and Bernstein, 2009), TA-SPARQL (Rodríguez et al., 2009) and stSPARQL (Koubarakis and Kyzirakos, 2010) used RDF as a data model, extended with time annotations. The time-related metadata in these approaches is associated to a named graph, and indexed so that time-based queries can be posed to the dataset. Temporal entailment and the notion of temporal RDF graphs is mostly based on the work of Gutierrez et al. (2007). These approaches, however, do not target continuous query processing but focus on time-based queries provided as SPARQL extensions, compatible with a traditional stored RDF model.

Subsequent extensions, oriented towards streaming processing, introduce the concept of stream graphs, which act like normal graphs but, because they hold timestamped triples, window operators can be applied to them. Streaming SPARQL (Bolles et al., 2008),

### 2.3. Extensions for RDF Stream Querying

---

C-SPARQL (Barbieri and Della Valle, 2010) and CQELS (Le-Phuoc et al., 2011a) incorporate this idea, although with some differences in syntax and semantics, as we will see later.

The extended grammar of Streaming SPARQL basically consists in adding the capability of defining windows (time or triple-based) over RDF stream triples. While Streaming SPARQL modifies the original semantics of SPARQL making it time-aware, C-SPARQL and CQELS rely on the existing mapping-based semantics of SPARQL, but add the specific definitions for windows (analogous to CQL for the relational streams). The minor syntactic differences can be seen in the following examples. For Streaming SPARQL, Listing 2.1 obtains the sensor temperature values sensed in the latest 10 minutes, every minute.

---

```
PREFIX fire:<http://www.sensorgrid4env.eu/ontologies/fireDetection#>
SELECT ?sensor ?temperature
FROM STREAM <http://sensorgrid4env.eu/data/Sensors.rdf>
WINDOW RANGE 10 MINUTE SLIDE 1 MINUTE
WHERE {
    ?sensor fire:hasTemperatureMeasurement ?temperature .
}
```

---

**Listing 2.1:** Streaming SPARQL query retrieving the latest 10 minutes of temperature values.

Notice that the stream is explicitly stated using the `STREAM` keyword, and windows are defined in the `FROM` construct, using the `WINDOW` keyword. The `SLIDE` parameter can be specified to indicate the frequency of the window creation.

C-SPARQL operators, apart from windows, include aggregate functions, and also allow combining static and streaming knowledge and multiple streams. The following example of a C-SPARQL query in Listing 2.2 obtains the temperature average of the values sensed in the last 10 minutes:

---

```
REGISTER QUERY AvgTempreature AS
PREFIX fire:<http://www.sensorgrid4env.eu/ontologies/fireDetection#>
SELECT DISTINCT ?sensor ?average
FROM STREAM <http://sensorgrid4env.eu/data/temperatures.trdf> [RANGE 10m STEP 1m]
WHERE {
    ?sensor fire:hasTemperatureMeasurement ?temperature .
}
AGGREGATE {(?average, AVG, {?temperature})}
```

---

**Listing 2.2:** C-SPARQL query retrieving the latest 10 minutes of temperature average values.

In a first version, C-SPARQL aggregates followed their own syntax and an additive-based semantics, which was different from the summarization aggregates of SQL ([Barbieri and Della Valle, 2010](#)). In later years C-SPARQL renounced to this paradigm and has aligned to SPARQL 1.1, mainly for compatibility reasons, as we can see in [Listing 2.3](#).

---

```
SELECT ?sensor
  MAX(?temperature) as ?avgTemperature
FROM STREAM <http://semsorgrid4env.eu/data/temperatures.trdf> [RANGE 30m STEP 5m]
WHERE {
  ?sensor fire:hasTemperatureMeasurement ?temperature .
}
GROUP BY { ?sensor }
```

---

**Listing 2.3:** C-SPARQL query retrieving the maximum temperature in the latest 30 minutes.

CQELS implements a native RDF stream query engine from scratch, not relying on a DSMS or CEP for managing the streams internally. The focus of this implementation is on the adaptivity of streaming query operators and their ability to efficiently combine streaming and stored data.

---

```
SELECT ?locName ?locDesc
FROM NAMED <http://deri.org/floorplan/>
WHERE {
  STREAM<http://deri.org/streams/rfid> [NOW]
  {?person lv:detectedAt ?loc}
  GRAPH <http://deri.org/floorplan/>
  {?loc lv:name ?locName.
    ?loc lv:desc ?locDesc }
  ?person foaf:name '$Name$'.
}
```

---

**Listing 2.4:** CQELS query retrieving the maximum temperature in the latest 30 minutes.

While the previous extensions to SPARQL were focused on windows and RDF streams, EP-SPARQL ([Anicic et al., 2011](#)) adopts a different perspective, oriented to complex pattern processing. Following the principles of complex event processing, EP-SPARQL extensions include sequencing and simultaneity operators. The SEQ operator intuitively denotes that two graph patterns are joined if one occurs after the other. The EQUALS operator between two patterns indicates that they are joined if they occur at the same time. The optional version of these operators, OPTIONALSEQ and EQUALSOPTIONAL, are also provided. As an example, the query in [Listing 2.5](#) obtains the companies with more than 50 percent drop in stock price, and the rating agency, if it previously downrated it.

### 2.3. Extensions for RDF Stream Querying

---

---

```
SELECT ?company ?ratingagency
WHERE {
  ?company downratedby ?ratingagency
}
OPTIONALSEQ {
  { ?company hasStockPrice ?price1 }
  SEQ { ?company hasStockPrice ?price2 }
}
FILTER (?price2 < ?price1 * 0.5)
```

---

**Listing 2.5:** EP-SPARQL sequence pattern example query.

Although EP-SPARQL lacks window operators, a similar behavior can be obtained by using filters with special time constraints.

Other recent approaches focused on event processing based on the Rete algorithm for pattern matching include Sparkwave ([Komazec et al., 2012](#)) and Instans ([Rinne et al., 2012](#)). However, Sparkwave requires a fixed RDF schema. Instans adds SPARQL update support to its features, but does not include windows on its query language because it claims that they may lead to inconsistent results (duplicates, missing triples because they fall outside of a window). This is not usually the case, as triples are normally not assumed to represent events on their own.

#### 2.3.2 Publishing Data Streams as Linked Data

In addition to querying RDF streams using SPARQL extensions, several systems have been devised to provide access to data streams on the Web in the form of Linked Data. The principles for publishing Linked Data provide simple guidelines for this task, namely (i) Use URIs as names for things, (ii) Use HTTP URIs to enable lookups, (iii) Provide useful information (RDF) when dereferencing URIs, (iv) Include links to other URIs ([Berners-Lee, 2006](#)).

In particular, we focus on sensor data publication in this section, as a specific type of streaming data. Early approaches, including the architectures described in ([Lewis et al., 2006](#)) and ([Huang and Javed, 2008](#)), rely on bulk-import operations that transform the sensor data into an RDF representation that can be queried using SPARQL in memory, lacking scalability and real-time querying capabilities.

In ([Wei and Barnaghi, 2009](#)) the authors describe preliminary work about annotat-

ing sensor data with RDF, using rules to deduce new knowledge. Semantic annotations are also considered for the specific task of adding new sensors to observation services in (Bröring et al., 2009). This work points out the challenges of dynamically registering sensors, including grounding features to defined entities, to temporal, spatial context. In (Jeung et al., 2010), the authors describe a metadata management framework based on Semantic Wiki technology to store distributed sensor metadata. The metadata is available through SPARQL to external services, including the system’s sensor data engine GSN, that uses this interface to compute distributed joins of data and metadata on its queries.

In (Babitski et al., 2009) a semantic annotation and integration architecture for OGC-compliant sensor services is presented. The approach follows the OGC-sensor Web enablement initiative, and exploits semantic discovery of sensor services using annotations. In (Henson et al., 2009) an SOS service with semantic annotations on sensor data is defined. The approach consists in adding annotations, i.e. embed terminology from an ontology in the XML O&M and SensorML documents of OGC SWE, using either XLink or the SWE swe:definition attribute for that purpose. In a different approach, the framework presented in (Le-Phuoc et al., 2010) provides sensor data readings annotated with metadata from the Linked Data Cloud.

While all these systems allow publishing sensor data on the web, they either lack streaming capabilities in their query languages or service interfaces, or provide little or no explicit semantics in the data, as in the RDF and SPARQL extensions we saw in Section 2.3.1.

## 2.4 Sensor Ontology Modeling

Ontologies provide a formal, usable and extensible model that is suitable for representing information, in our case sensor data, at different levels of abstraction and with rich semantic descriptions that can be used for searching and reasoning (Corcho and García-Castro, 2010). Moreover in a highly heterogeneous setting, using standards and widely adopted vocabularies facilitates the tasks of publishing, searching and sharing the data.

In the previous sections we explored the challenges of querying and managing sensor data, and how semantics can be used to enrich this type of data, as well as allowing interoperability and reasoning to varying extents. In all cases, it is necessary to provide semantic models and vocabularies for this type of data. In this section we explore some of the most relevant works related to the use of ontologies for modeling sensor devices, sys-

tems, platforms, capabilities, metadata, measurements, observations and other related concepts.

### 2.4.1 Ontologies for Sensor Data and Metadata

The task of modeling sensor data and metadata with ontologies has been addressed by in recent years, and several sensor ontologies have been proposed, some of them focused on sensor descriptions, and others in sensor measurements (Compton et al., 2009a).

Sensor metadata is typically static -i.e. it seldom changes over time- and includes information such as sensor types, descriptions, procedures, technical capabilities, value ranges, etc. This semantic metadata can be used for several purposes, including enriched search, discovery of datasets, or sensor configuration. On the other hand sensor observation data is characterized by higher data volumes and continuous updates, and includes the values captured by the sensors, for a certain feature.

Early ontology proposals for describing wireless sensors were presented in (Avancha et al., 2004; Eid et al., 2007) and others, as summarized in Compton et al. (2009a). However in these approaches the focus was on sensor meta information, while the description of observation was overlooked. Besides, some of these approaches lack ontology design best practices of reuse and alignment with standards and reference ontologies. Newer models, including the OntoSensor ontology (Russomanno et al., 2005), are based on the concepts defined in the OGC<sup>1</sup> SensorML<sup>2</sup> standard as a basis. The OGC has led a task-force for standardizing sensor metadata, although focused on a syntactic representation that lacks explicitly stated semantics. The sensor description classes and properties of OntoSensor, include the concepts of Sensor, Sensor Capabilities, Platform, Observable and Measurand. It also has extensions to the SUMO upper ontology, and service extensions have been proposed for it (Kim et al., 2008). It is one of most complete ontologies in terms of coverage of sensor metadata, but it does not take into account observation data, not even following the OGC Observations and Measurements (O&M) standard<sup>3</sup>. Similar proposals such as the SWAMO ontology (Witt et al., 2008), which is also focused on the description of sensors, are based on the concepts defined in the OGC SensorML standard.

A more recent proposal of (Barnaghi et al., 2009), proposes extending ontology models to also represent observations captured by sensor networks, following the OGC Observations and Measurements (O&M) standard. The ontology also covers features of sensor

---

<sup>1</sup>Open Geospatial Consortium [www.opengeospatial.org/](http://www.opengeospatial.org/)

<sup>2</sup>SensorML: <http://www.opengeospatial.org/standards/sensorml>

<sup>3</sup>OGC Observations & Measurements: <http://www.opengeospatial.org/standards/om>



nodes such as power supply, operation modes and typical values. However it does not include deployment or configuration concepts, and moreover it does not follow the design principles of modularization and reuse of existing vocabularies.

In (Eid et al., 2007) the authors describe the sensor data and hierarchy ontology SDO, which is divided in different interlinked modules, is defined as an extension to the SUMO ontology<sup>1</sup>. Extensions to such upper level ontologies are interesting from the point of view of integration with other vocabularies and potential reuse. The MMI (Marine Metadata Interoperability) device ontology<sup>2</sup> is another initiative to model sensor devices, and although centered on the marine domain can be used in other contexts. Even though it has been mapped to other RDF vocabularies in the MMI community, it lacks modeling for data and observations.

The CSIRO sensor ontology is presented in (Compton et al., 2009b), and apart from sensor descriptions and properties, includes the ability to represent sensor composition and components, making it possible to represent virtual sensors as well. However, data and observations concepts are not fully detailed in the ontology. These ontologies have also been used to define and specify complex events and actions that run on an event processing engine (Taylor and Leidinger, 2011).

These and other ontologies are reviewed in (Compton et al., 2009a). The evaluation covers the ability to describe: sensor types and hierarchies, sensor data and measurements, sensor systems, composition and structure, etc., and identifies shortcomings in all ontologies, particularly regarding the ability to describe configurations, response models or operating conditions to a satisfying level.

### 2.4.2 The SSN Ontology

Based on all these previous approaches, and through the W3C SSN-XG group<sup>3</sup>, the semantic web and sensor network communities have made an effort to provide a domain independent ontology, generic enough to adapt to different use-cases, and compatible with the OGC standards at the sensor level (SensorML) and observation level (O&M). The result, the SSN ontology (Compton et al., 2012), is based on the stimulus-sensor-observation design pattern (Janowicz and Compton, 2010) and the OGC standards. The SSN ontology is aligned to the upper level DOLCE Ultra Lite ontology, to enhance compatibility and matching with other existing ontologies, and as a W3C group initiative,

---

<sup>1</sup>SUMO ontology here

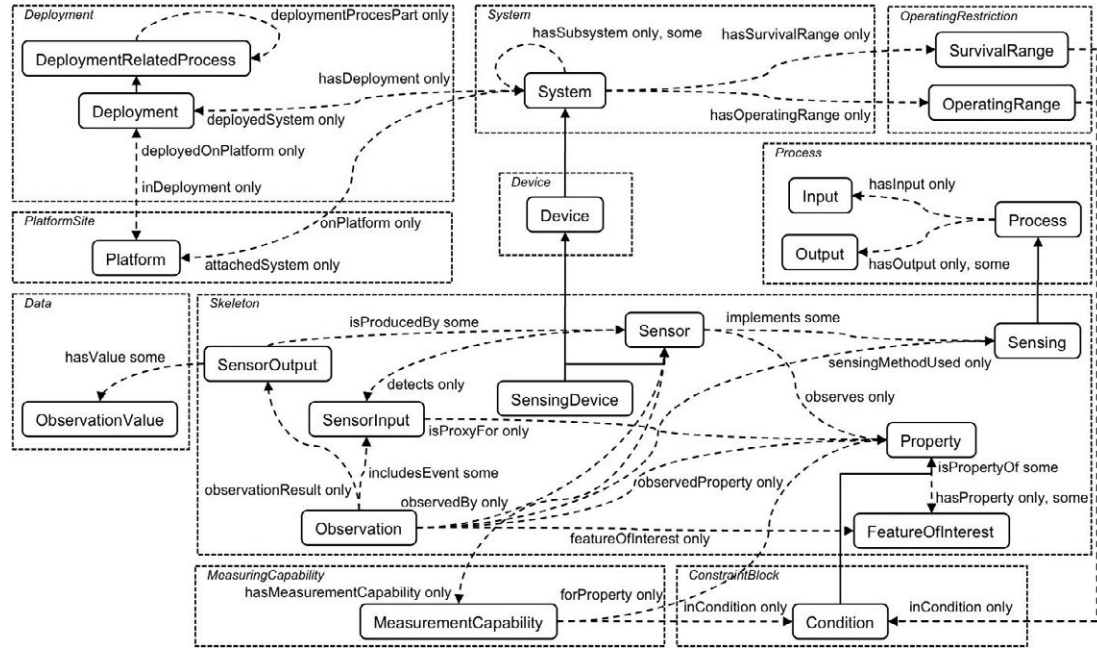
<sup>2</sup><https://marinemetadata.org/community/teams/ontdevices>

<sup>3</sup>[www.w3.org/2005/Incubator/ssn](http://www.w3.org/2005/Incubator/ssn)

has followed a collaborative development process.

The SSN ontology (see Figure 2.7) can be used to describe sensors, and how they function and process the external stimuli. Besides, it can be centered on the observed data, and its associated metadata, or in the systems and platforms of a sensing deployment. These different perspectives are reflected in the SSN modules. The core *Skeleton* module includes concepts used in almost every use-case, such as sensors, the observations that they produce, including observed properties of certain features of interest.

The *Data* module is focused on the data values recorded by the sensors, while the *Device* and *Measuring Capability* modules deal with technical device properties. Form the point of view of the *Deployment* and *System* modules, we get a view of the organization and composition of sensors and devices. Depending on the use-case and requirements, some of these modules may be needed and others may not, but in general the Skeleton concepts are used in most applications (Compton et al., 2012).



**Figure 2.7:** SSN ontology main classes and modules.

As an example, consider a wind-monitor sensor in a weather station deployed at a field site. The sensor is capable of measuring the wind speed on its specific location. Suppose that another sensor attached at the same station reports air temperature every 10 minutes. Using the SSN ontology we can represent both as instances of `ssn:Sensor`<sup>1</sup>,

<sup>1</sup>We use URI prefixes in the inline text for clarity, e.g. `ssn`.

each with a different URI (Listing 2.6). Using the `ssn:onPlatform` property we can link each sensor to a weather station platform, e.g. Wannengrat7 platform, where both are installed. The platform is geo-spatially located, using the WGS84 vocabulary<sup>1</sup>. In the example, the location coordinates (latitude and longitude) of the platform are provided, as a geographical point. We can also include other information such as a responsible person, initial date of the deployment, etc.

---

```
swissex:sensorWind1
  rdf:type ssn:Sensor;
  ssn:onPlatform swissex:platformWannengrat7;
  ssn:observes cf-property:wind_speed.
swissex:sensorTemp1
  rdf:type ssn:Sensor;
  ssn:onPlatform swissex:platformWannengrat7;
  ssn:observes cf-property:air_temperature.
swissex:platformWannengrat7
  ngeo:hasGeometry [ rdf:type wgs84:Point;
                    wgs84:lat  "46.8037166";
                    wgs84:long "9.7780305" ]].
```

---

**Listing 2.6:** Representation of sensors on a platform station and its location using the SSN ontology

Finally, in the example we use the `ssn:observes` property to indicate the type of observed property. Because the SSN ontology is domain-independent, the actual values of the observed property are to be taken from specific vocabularies, such as the Climate & Forecast properties ontology<sup>2</sup> in this case. Thus, for the wind sensor we use the `cf:property:wind_speed` term to represent the wind speed observed property.

In terms of the SSN ontology both the wind and temperature measurements can be seen as observations, each of them with a different feature of interest (wind and air), and each referring to a different property (speed and temperature). In the SSN ontology, instances of the Observation class represent such observations, and are linked to a certain feature instance through a `featureOfInterest` property (see Figure 2.7). Similarly the `observedProperty` links to an instance of a property, such as speed. Evidently more information about the observation can be recorded, including units, accuracy, noise, failures, etc. Notice that the process of ontology modeling requires reuse and combination of the SSN ontology and domain-specific ontologies.

---

<sup>1</sup>Basic Geo WGS84 Vocabulary: <http://www.w3.org/2003/01/geo/>

<sup>2</sup>Climate & Forecast property ontology: [www.w3.org/2005/Incubator/ssn/ssnx/cf/cf-property.html](http://www.w3.org/2005/Incubator/ssn/ssnx/cf/cf-property.html)

## 2.5 Sensor Data Approximation and Classification

As a final section for this state of the art, we focus on the different existing techniques for data processing that are used as a basis for more advanced tasks such as data analysis, efficient storage, and pattern mining. Raw data captured by sensors may be useless unless it is placed in context, and usually requires pre-processing before being published or being available for querying. We address two main tasks in this section: approximation and compression of raw data, and classification of sensor data.

The former is a basic task used for efficient data storage and for generating a compact representation for further processing. It is relevant because it provides more useful representation for the data to be published as RDF or made available through SPARQL queries. The latter is a key task for identifying relevant metadata and make sense out of, otherwise useless, raw values.

### 2.5.1 Data Approximation and Compression

Sensor data is typically represented as time series, which are usually too large to be processed by analysis, mining and classification algorithms. There are a number of methods for sensor data approximation, that can be used to extract relevant features, patterns or for data compression. High level representations reduce the dimensionality of the data, in order to reduce the complexity of indexing and comparison algorithms, using different techniques.

In general, data compression aims at representing raw data values as a set of functions (Sathe et al., 2012). A raw data values series  $D = \{d_1, \dots, d_n\}$  can be split into a sequence of sub-series  $D_i = \{d_{1_i}, \dots, d_{k_i}\}$ , where each data point is a tuple  $(t, v)$ , with  $t$  representing time and  $v$  the measured value. The approximation is given by  $F = \{f_i \mid f_i \text{ approximates } D_i\}$ , i.e. a set of functions, each of which approximates a sub-series  $D_i$ .

The approximation is usually associated to an error calculated with respect to the raw data. This error is expected to be minimized by the compression algorithms. Approximations are useful for different purposes, for example efficient storage, avoiding storing all data values but only the function parameters instead. It is also useful for data analysis algorithms, which would not be computationally feasible for very large amounts of raw data.

We will now see the most common techniques for approximating time series, including segmentation, linear approximations, symbolization, and others.

**Time Series Segmentation.** One of the simplest functions for data approximation is linear segmentation. The original series  $D$  is represented as a sequence of segments, each representing a sub-series  $D_i$  as a linear function  $ax + b$ . However simple this technique may look like, there exist many alternative approaches for segmenting the original series, such that the approximation error is minimized.

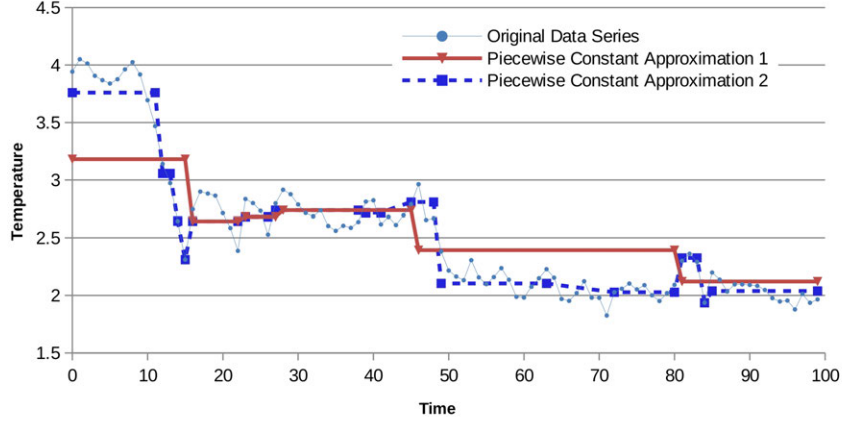
Time series segmentation may follow four main approaches (Keogh et al., 2004):

- **Top-down:** It recursively divides the series in two segments, until all segments are approximated within a given maximum error boundary.
- **Bottom-up:** Basic segments are created initially, and then the contiguous segments are merged if their fusion increases the error the least. These segments are merged until no segment merge is possible without reaching a maximum error boundary.
- **Sliding window:** A segment grows until it reaches some error boundary. Then, for the next data values existing after the new segment, the same process is repeated. Each of this non-overlapping sets of points constitutes a window.
- **Bottom-up window:** A combination of the bottom-up approach and the sliding window approach. It performs the bottom-up algorithm within a buffer of preset size. Hence it does not require the whole dataset offline for computation.

**Linear Approximations.** Piecewise linear approximations have been widely used as time series representation, because of the simplicity of computation, while preserving acceptable error margins. The simplest linear representation uses constant segments for representing a subset of data points.

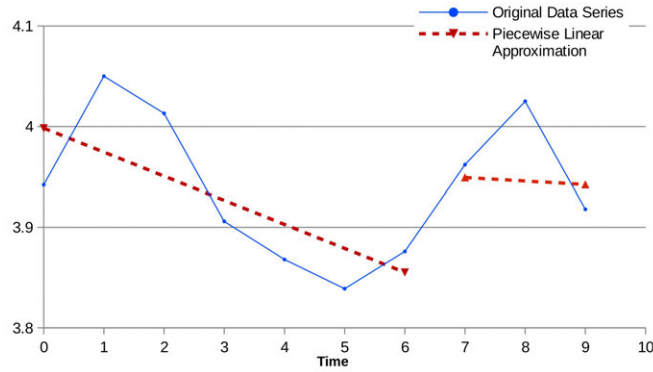
For instance in Figure 2.8, we use PAA (Keogh et al., 2001) constant segments to represent the original data (e.g. the average of the points in that interval). We show how the original data series can be approximated with constant segments, with different number of segments. Obviously, if we use more segments, the series is better approximated, but we require to store more information. These techniques (PCA (Keogh et al., 2004), APCA (Chakrabarti et al., 2002), etc.) are useful for data compression but do not tell us much about the trends in sensor data, e.g. if it has sudden peaks or other interesting variations in the series.

Alternatively if we approximate using linear segments (Piecewise Linear Representation, PLR), we can already see the trends of the time series by observing the angles that they form. For instance the series in Figure 2.9, uses 2 segments to represent the original 10 data points. Notice that the number of points for a segment can be variable



**Figure 2.8:** Constant piecewise approximation.

(adaptive approximations). Algorithms such as (Keogh et al., 2004; Keogh and Smyth, 1997) have been devised for this purpose.



**Figure 2.9:** Piecewise linear approximation.

Generally simple to compute, either in batch mode and online using sliding window algorithms, these methods offer accurate approximations of the original data. These representations have been widely used for tasks including similarity search, fuzzy queries, dynamic time warping, clustering and classification (Keogh et al., 2004).

**Symbolization.** While the mentioned approximations reduce dimensionality, some approaches introduce a further step that consists in symbolization of the time series. These techniques, such as SAX (Lin et al., 2007), have shown to be space and time efficient for indexing, classification and clustering, and also for additional tasks such as motif dis-

covery and visualization ([Kasetty et al., 2008](#)). After applying a PAA constant approximation over the original time series, SAX divides the amplitude space in sections, and assigns a symbol to each one. The division is made such that the probability of a data point being assigned to any symbol is the same. These symbolizations can be used to compute distance measures that help in classification and clustering tasks ([Lin et al., 2007](#)). Other works have considered also the slopes of linear approximations such as the STS distance ([Möller-Levet et al., 2003](#)) for clustering time series.

**Alternative Approximation Techniques.** Other alternative and/or complementary approaches have been proposed in the literature, and although we mention them briefly, a more detailed overview can be found in ([Ding et al., 2008](#); [Sathe et al., 2012](#)).

- **Fourier Transform:** These approaches use a feature extraction technique on time series, using the fact that a signal can be described as a combination of sine and cosine functions. Using a transformation such as Discrete Fourier Transform (DFT), only the coefficients of the approximation are needed to be stored ([Agrawal et al., 1993](#)). These coefficients can be indexed for search, comparison and distance computation, for which many extensions have been proposed ([Faloutsos et al., 1994](#); [Korn et al., 1997](#)).
- **Wavelets:** Using the Discrete Wavelet Transform (DWT) a series can be represented as a combination of other types of functions apart from sine and cosine ([Kin-Pong Chan et al., 2003](#)). With DWT it is possible to capture local features and not only local features of the time series, and is a well-known compression technique used in other domains including imaging and audio.
- **Correlation-aware compression:** Correlations among different time series may be exploited for higher compression. Intuitively, highly correlated series may present redundancies that can be used by compression approaches such as GAMPS ([Gandhi et al., 2009](#)) or SBR ([Deligiannakis et al., 2004](#)) to group and jointly compress them.
- **Multi-model compression:** In some scenarios, using the same model for the whole time series is not adequate. Different segments of the data may have completely different characteristics, rates or value ranges. Dynamic approaches such as ([Papaiannou et al., 2011](#)) consider burstiness, data rates, ranges, for choosing the most appropriate model for a particular segment of the stream, using the compression ratio as the main comparison criterion.



A comprehensive comparison and evaluation of these representation approaches, as well as distance measures has been presented in (Ding et al., 2008), showing that despite the variety of methods available, they have similar efficiency in terms of Tightness of Lower Bounds.

### 2.5.2 Time Series Classification

There is a large body of research aimed at analyzing and mining time series data, for different purposes. Particularly, for the task of classification, different techniques such as decision trees, neural networks and Bayesian classifiers have been used (Xing et al., 2010). The classification problem can be stated as follows: Given an unlabeled time series  $s$ , assign it to a class label  $l$ , from a set of predefined class labels  $L$ .

Classification approaches have been grouped into the following three categories according to Xing et al. (2010): distance-based, feature-based and model-based.

**Distance-based.** This approach uses distance functions to compare the similarity of two time series. These distance measures can be used to perform a classification using an existing method such as k-nearest neighbors (k-NN). Given a set of already classified series, and an unclassified series  $s$ , k-NN finds from this set the  $k$  closest series to  $s$ . The class label with most occurrences from these  $k$  series will be assigned to  $s$ . The  $k$  parameter can be tuned depending on the use case.

In this type of classification, choosing an appropriate distance function is key for obtaining results. Basic distance measures such as euclidean distance, may result surprisingly effective in many scenarios, while very simple to compute (Keogh and Kasetty, 2003). However, distance measures such as the euclidean, are limited to same-length series, and they only consider one-to-one matches in the time axis. Small misalignments may result in large distances that do not reflect the real similarity of the data values.

Distance measures with more elastic matching for the time axis, such as Dynamic Time Warping (DTW), have been proved effective for similarity matching (Ding et al., 2008). DTW consists in aligning two series flexibly, comparing a point of a series with a small buffer of other points on the other series. The best alignment is the one that results in smaller distance. The size of the buffer affects the efficiency of this method, while a large one may result in a better alignment, it is computationally more expensive. This distance measure has also been coupled with k-NN classifiers, proving an effective combination for a number of time series classification problems (Geurts, 2001; Xi et al., 2006). Nevertheless, k-NN requires an important number of preclassified series to work



well. Otherwise, the comparison set of series may not be representative enough. For example we may have a large number of preclassified series of label  $a$ , but few labeled as  $b$ . Then the classifier may not have sufficient samples to identify future  $b$  series.

These techniques have space and time computation limitations in some scenarios, and some optimizations including numerosity reduction have been implemented (Xi et al., 2006). Moreover, although effective in many scenarios, they usually offer little explanation on why a series belongs to a particular class (Ye and Keogh, 2009).

**Feature-based.** Feature-based approaches try to find properties that are representative of a type of series, in order to classify them. Most of these approaches use a high level representation e.g. symbolization or discretization methods, before extracting the features (Xing et al., 2010) while others work extracting representative sub-sequences.

SVM (Support Vector Machine) (Cristianini and Shawe-Taylor, 2000) uses a kernel function to transform a series into a feature vector. Then in this feature space, SVM finds the maximum-margin hyperplane to separate the classes that those feature vectors represent. There are variants of this approach, including choosing different kernels (Osuna et al., 1997). However, SVM classification often suffers from little interpretability of the classification for users. Other feature extraction techniques include FeatureMine (Lesh et al., 1999), that uses heuristics to identify sub-sequences in the series, such that they are representative of one class, frequent, and that avoid redundancy.

A simpler feature based extraction consists in representing every series element as a feature on a vector. However this results very inefficient in large series, and unfeasible in most sensor time series, because these are mostly numerical. For this reason, several numerosity reduction techniques have been used to transform the series data into a feature vector. These techniques include the approximations in Section 2.5.1. Other techniques intend to find representative portions of a series, that can be associated with a certain class. Then, if these portions are found in other series, then it can be classified in the corresponding class. Shapelets (Ye and Keogh, 2009) are one example of such representative sub-sequences, such that they separate the training set in two different parts, based on the distance to the shapelet. Although the search space for shapelets is huge, there are techniques that reduce it, for instance avoiding computing all distances from a candidate to a time series (early abandon), and calculating an upper bound for the information gain (entropy pruning).

**Model-based.** Model based classification is based on the assumption that classification classes can be generated by a certain model. these models take a number of parameters,

such that they are able to generate the sequences of the classes. Then, the goal is to learn these parameters from a training dataset, so that new time series can be assigned the model that is able to generate the sequence more precisely. Examples of such model-based classifiers include Naive Bayes, Markov Model and Hidden Markov Model (Xing et al., 2010).

### 2.5.3 Characterizing Unknown Sensor Data

The approximation and classification techniques described above, have been applied in different use cases, and in particular for sensor data processing, pattern recognition and mining in general. However, in the emerging Sensor Web sensor data sources may be unknown, in the sense of lack of metadata and provenance information (Corcho and García-Castro, 2010). These sources might not be described enough, or provide insufficient metadata to be usable at all.

Some of the key pieces of information about data sources in the Sensor Web, is the nature or type of observation being recorded. In terms of the SSN ontology or the OGC O&M models, this is the type of the *observed property*, e.g. relative humidity, percentage of carbon dioxide, barometric pressure, etc. Another essential information is the *feature of interest*, e.g. the air at some location, the water at some river, etc. While in closed systems this metadata is provided manually by the data providers, in the Web, there are no guarantees for its accuracy or its provision at all. While there is a large body of work on time series classification as we saw in Section 6.3.2, and stream mining (Gaber et al., 2005), few approaches tried to use these techniques for characterizing sensor data sources with semantically-aware models.

Some approaches enable metadata enrichment of data streams, such as in (Legeay et al., 2007), producing XML documents. Metadata enrichment for data fusion has also been covered, for instance in (De Mel et al., 2011), although it covers only the use of existing metadata as an input, and not using the sensor observations as well. A conceptual framework for metadata enrichment has also been proposed in (Moraru and Mladenice, 2012), which is targeted at automatically producing a semantic sensor data catalog. Although this model is comprehensive and explicitly looks for instances of essential metadata such as the observed property and feature of interest, does not provide hints on how this can be implemented, e.g. using mining techniques, etc. Finally, the work of (Barnaghi et al., 2012), describes a technique for generating perception metadata from raw observations. It relies on the SAX symbolization for finding data patterns that can be

later used to be matched to perception abstractions.

## 2.6 Discussion

In this chapter we described the most relevant approaches in the different areas that this thesis is built upon.

Concerning the first challenge of **querying streaming data** through ontology-based languages, we have first explored the different processing models and languages in the streaming data and event processing related work. (Section 2.1). We evidenced the advances in this area, considering extensions for the relational model, and the inclusion of continuous query processing paradigms. In addition, many of the resulting systems deal with scalability and performance constraints, typical of applications of this technology. However, these approaches lead to the emergence of streaming data silos, each running a particular data model, technology, accessible through a certain query language. Although there are technological initiatives for a common standard in certain areas, such as the OGC-SWE, these are based on syntactic agreements of format, and lack any explicit semantics. Moreover, this model is closed and prevents easily extending the model or aligning it to another.

Although **ontology-based access and querying** addresses these challenges, as seen in Section 2.2, the existing approaches are targeted to static stored data only, disregarding streaming data sources. As an alternative, several proposals have surfaced concurrently to this thesis, with the goal of **extending RDF and SPARQL for streaming data processing** (Section 2.3). As we have seen, these proposals in some cases implement a query processor from scratch (e.g. CQELS) or are implemented on top of a DSMS (e.g. C-SPARQL). Nevertheless, none of these approaches is flexible enough to provide SPARQL streaming query access to a variety of existing DSMS, CEP or even sensor middleware.

As we have seen, some of these systems have equivalent streaming operators, and sensor middleware systems provide only limited service interfaces (e.g. GSN, Cosm), which are equivalent to simple query interfaces. To our knowledge, none of the previous approaches allow querying such a wide range of stream processors, using RDF and SPARQL extensions.

Despite the differences in these highly heterogeneous data sources, in terms of tech-

nology and data structure, we have seen that it is possible to use ontologies as a rich model that is reusable and can be adapted to domain-specific scenarios. Specifically for sensor data, we have explored the existing alternatives for **both sensor and observation ontology modeling** in Section 2.4. However, in the context of the Sensor Web, where metadata may be incomplete, faulty or simply missing, these ontological representations need mechanisms that allow their enrichment. In particular, we have seen methods for approximating and classifying time series data, that can help **characterizing raw measurements**. Although these principles have been useful for pattern extraction and querying-by-example, they have not been used for deriving semantic metadata that characterizes a type of sensor measurement.



## Chapter 3

# Hypotheses and Contributions

In this chapter we state the work hypotheses that we want to validate, under certain assumptions and limitations. From these hypothesis, we draw the thesis objectives and discuss the associated conceptual, methodological and technological contributions proposed in this work.

### 3.1 Problem Statement

This work aims at providing theoretical foundations and technical solutions for accessing sensor data streams through semantic technologies. From the analysis of the state of the art in Chapter 2, we have identified two main problems that are specifically addressed by this work:

- P1. Accessing streaming sensor data with continuous queries using ontology models to represent the sensor observations.
- P2. Characterizing sensor data using the recorded time series, capturing the semantic metadata of the sensor observations.

For the first problem, the following research questions are addressed in this work:

- *How can we use ontology models to query real-time data streams originated from sensors?* Ontologies have been used successfully to query heterogeneous static relational data sources. However, sensor streaming data requirements differ from static ones, as they require continuous and recency-aware query processing and data delivery. Our goal is to explore ontology-based access to data streams.

- *Can we access heterogeneous DSMSs, CEPs and sensor data middleware effectively using ontologies as a data model?* Considering the widespread adoption of DSMSs, CEPs and web middleware for managing sensor data, and the heterogeneity of the data access mechanisms that they provide, it is needed to provide means of querying these systems under a coherent data abstraction. Using ontologies as such a model, we aim at establishing a method for rewriting queries from ontologies to data stream schemas, and translating the results accordingly.
- *Can we develop SPARQL extensions for streams suitable for querying streaming data coming from DSMSs and CEPs?* SPARQL is the de-facto standard for querying RDF data. In case of using DSMSs and CEPs as underlying processing engines, SPARQL language extensions for recency-aware and continuous query operators need to be defined in terms of operators available in these systems.

For the second problem, regarding the characterization of semantic properties of sensor data, we consider the following research questions:

- *What data representations are suitable for extracting sensor data features that characterize a set of data streams?* Different data representations and dimensionality reduction techniques can be used to represent sensor time series. Our goal is to explore and propose representations that are useful for extraction of characteristic features of sensor observations.
- *Which data mining classification and mining techniques can be used for data characterization?* Using an appropriate representation, distance measures and classification techniques, we aim at proposing data analysis and mining methods that allow classifying sensor data from given time series.
- *How can we represent semantic properties of sensor data using ontologies?* Using the extracted metadata information, the existing sensor metadata can be enriched with semantic properties that correspond to the metadata learned with the characterization techniques.

Both research problems deal with sensor data from different perspectives. For the first one we explore how we can use semantic technologies to query the observations that they produce. For the second we focus on the enrichment of its metadata by extracting and mining information from raw data observations.

## 3.2 Hypotheses

We formalize the work hypothesis that follow from the research challenges discussed in Section 3.1. For the research problem P1, we elaborated the following hypotheses:

**H1.** Sensor streaming data can be treated as instances of concepts from a comprehensive ontology model, such as a combination of the SSN ontology with domain-specific ontologies.

**H2.** SPARQL queries can be extended with streaming operators, and used to continuously query streaming data, in terms of the ontology concepts and properties.

**H3.** Ontology-based queries to streaming data can be rewritten into queries in terms of relational and event streams, and the query results can be returned as instances in terms of the ontologies. The use of existing relational-to-RDF (RDB2RDF) mappings can be extended to streaming data for this purpose.

**H4.** Ontology-based queries for streams can be expressed as abstract expressions that can be transformed into queries for relational streams and complex event processors, or sensor middleware service requests.

**H5.** Ontology-based query rewriting for streams can be applied for both push and pull data delivery, incurring in an acceptable overhead in query latency.

For the second research problem P2, we identified the following hypothesis:

**H6.** Sensor data series can be analyzed in order to find patterns in them, which characterize its type and makes it recognizable among other types of series.

**H7.** Sensor data time series representations can be analyzed, so that semantic properties such as the type of data, can be learned using mining clustering and classification techniques, with acceptable precision.

### 3.2.1 Assumptions

The hypothesis are stated considering the following assumptions:



**A1.** Ontological models that represent the domain of interest are available or can be developed using state-of-the-art ontology engineering practices, and are integrated with standard sensor network ontologies, like the SSN ontology.

**A2.** Data stream management systems and complex event processors that will execute the rewritten queries, are able to compute continuous queries over data streams in real time.

**A3.** Sensor and streaming data providers already host and manage their data with DSMSS, CEPs or web based data repositories.

**A4.** Mappings from relational or event streams to ontologies, can be defined by domain experts and ontology engineers with or without helper tools.

**A5.** For sensor data characterization, a representative and sufficiently accurate training set is assumed to be available.

### 3.2.2 Limitations

The goals of this PhD thesis are bound to the following limitations:

**L1.** The ontology-based query rewriting is restricted to medium sampling rates of sensor data, typical of domains such as environmental monitoring.

**L2.** The expressivity of ontology-based queries that can be executed over a given data stream is limited to the expressivity of the queries supported by underlying DSMSS, CEPs or web middleware services.

**L3.** For arbitrary streaming data sources with custom query and data formats, adapters must be implemented in order to use the proposed rewriting principles.

**L4.** The approach for ontology-based queries over streams only provides simple entailment<sup>1</sup>, but other query rewriting techniques can be plugged in to support other entailment regimes.

---

<sup>1</sup>SPARQL entailment regimes:<http://www.w3.org/TR/sparql11-entailment>

**L5.** Arbitrarily noisy sensor data series with hardly recognizable patterns cannot be used for accurate characterization.

**L6.** Low number of sensor time series in a training set produce low precision results for data characterization.

**L7.** Data characterization is not computed in real-time, but offline.

## 3.3 Contributions

The main contribution of this work is a novel set of methods and techniques for accessing and querying streaming sensor data from heterogeneous sources, using ontologies to represent both sensor data and metadata. This body of work stems from the investigation of ontology-based query rewriting techniques, extensions for streaming operators and windows and the use of declarative mappings to capture data from non-ontological sources. This contribution addresses the first research problem in Section 3.1, and its specific conceptual contributions are:

- The formalization of SPARQL with streaming extensions (SPARQL<sub>Stream</sub>) in terms of relational streaming algebra expressions.
- A method to rewrite SPARQL<sub>Stream</sub> queries as algebra expressions using declarative mappings that relate ontological schemas to stream schemas.
- A method to translate relational or event stream results to SPARQL query results or triples.

We also provide technological contributions that enable ontology-based access to data streams, namely:

- An implementation of SPARQL<sub>Stream</sub> for continuous queries, and pluggable to an underlying stream management system.
- A query rewriting and data translation implementation, for DSMS and CEP, and sensor web middleware, using declarative mappings written in R2RML. In particular we provide sample implementations for SNEE, Esper, GSN and Cosm.
- An implementation of pull and push data delivery mechanisms for querying streaming data using the ontology-based query rewriting approach.

For the second research problem of Section 3.1, related to the characterization of sensor data, our contributions are:

- A representation of sensor time series that captures gradient information that is useful to characterize types of sensor data.
- A method for classifying sensor time series and determining the type of data, using data mining techniques.
- A method for extracting sensor metadata features from the time series and representing them as semantic properties.

A technical contribution for this problem, is an implementation of the semantic metadata extraction approach, for potentially unknown sensor data sources.

## Chapter 4

# A SPARQL Extension for Querying Data Streams

In order to allow ontology-based access to data streams, we require a general data model able to represent both the conceptualizations present in ontologies, and the streaming and temporal aspects of the data. This model also needs a corresponding query language able to cope with these features. As we have seen in Section 2.3, RDF streams have been introduced as a suitable data model in the literature, while extensions for SPARQL have been proposed as query languages for such type of data. However, all of these extensions assume that streams are natively represented in RDF, while in many scenarios, underlying DSMS, CEP engines and streaming data middleware are used to manage and store data dynamically. We propose SPARQL<sub>Stream</sub>, a query language that extends SPARQL for querying streaming RDF data. SPARQL<sub>Stream</sub> allows querying and exposing the data in terms of the high-level concepts of an ontology, while internally the data is managed using structures or schemas that are understandable by the underlying DSMS, CEP engine or streaming data middleware<sup>1</sup>.

In this chapter we enumerate key features for ontology-based streaming data querying, then describe the RDF stream model used by SPARQL<sub>Stream</sub>, and the syntax and semantics of the query language. In particular, we describe the semantics of the language in terms of relational stream algebra expressions. Afterwards, we discuss about query optimization techniques applicable to query rewriting of these queries. Finally we show how the query results are translated to triples or SPARQL variable bindings, closing the query interaction.

---

<sup>1</sup>We will use the term *streaming data sources* for referring to these three types of systems.

## 4.1 Features of a Semantically-enabled Streaming Query Language

In this section, we identify some of the most relevant features for querying streaming data sources, using ontologies as a data model. Considering the overlapping aspects of stream and event processing (Chandy et al., 2011), in the following we do not emphasize their differences but consider more general requirements that are applicable in both cases (see Section 2.1 for more details on both types of query processing).

**Representing Data with a Streaming Data Model.** Queries over data streams require a data model that is capable of dealing with dynamic and potentially unbounded online data, representing the evolution of some feature over time (Babcock et al., 2002). This notion contrasts with classical stored or static data that represents a snapshot of a certain feature in a given moment.

Time-aware data models can be as simple as classical tuples, where one of its attributes stores a timestamp. However, this may not be enough as there might be different time-based attributes representing different notions such as: observation-time, storage-time, sampling-time, etc. This may lead to ambiguity when constructing queries over this type of data models. Moreover, to enable ontology-based queries, we adopt a data model where clear explicit semantics can be encoded in concepts and properties. RDF is a successful and standardized model that complies to this need, so we need an **RDF-based model extended with a native abstraction of the notion of the change of the values over time.**

**Processing Continuous Queries.** The notion of continuous querying has been long studied in data stream processing (Arasu et al., 2006), and an analogous mechanism is needed for semantically-aware query languages, where events can be specified as graph patterns that are evaluated as the data changes with respect to time. Then, registered queries may also produce continuous results that describe new information in a dynamic way. Hence **a continuous query execution is needed to support processing live streaming data.**

**Formulating Declarative and Expressive Queries.** SQL for relational databases and SPARQL, for RDF, are declarative query languages that have been developed and standardized, and are being widely used in research and industry. These queries specify *what* data is requested and not *how* it is obtained. This allows a simple definition

of query requirements and hides internal -and possibly different- implementation and optimization details that are orthogonal to the expected answers.

Query expressiveness is characterized based on the operators available in the query language to extract the needed information. Extracting subsets of useful and meaningful information from the streaming data sources is one of the main goals of querying. In order to achieve this goal, a rich set of operators is required, such that it is possible to specify:

- Data filters, selecting the desired information by applying conditions, e.g. a data value higher than a given threshold.
- Projections, selecting only a subset of attributes
- Computed attributes, resulting from applying a function over other attributes, e.g. an arithmetic operation over observed data values.
- Aggregation, e.g. the average value of a certain observed property value.
- Sliding Windows, operating only over a subset of the data, limited by the window boundaries. For instance, selecting the latest 10 minutes of observation values.

These are some of the most common data operators required in streaming query processing, although we do not exclude others that may be useful in particular scenarios. However, these are minimal data selection requirements identified by the streaming and event processing communities ([Arasu et al., 2006](#); [Stonebraker et al., 2005](#)). Stream processing over an ontology-aware model such as RDF, is easily coupled with **declarative and expressive languages** to state query requirements. Although there are proposals such as CQL ([Arasu et al., 2006](#)), similar extensions are desirable for an RDF-based language as SPARQL.

**Integrating Streaming and Static Data.** Even though we are focused on streaming data and events, these are usually linked to contextual information contained in stored and static repositories. Moreover, in the case of ontology-based data, these links to stored data sources may require mixing different vocabularies in the query, representing different concepts, which may have a streaming nature or not. For example, a query may request streaming weather observations in terms of a climate ontology, while asking for its location in terms of a geo-location ontology in a stored dataset.

**Mixing these types of data sources in a single query, allows combining data, providing context, or comparing live data with historical information** ([Stonebraker et al., 2005](#)). Here we do not discuss the distributed or federated nature of the

data source, which can be reflected in the query as well (Acosta et al., 2011; Buil-Aranda et al., 2011), as it falls out of the scope of our work.

**Generating New Information.** Queries not only request for existing information but may also produce new one. For example in the case of events, a query may periodically detect the presence of a flood in a region, judging by the wind speed conditions and the sea wave height. For ontology-based queries, this new information can be represented in terms of high level concepts that correspond to a domain ontology. In subsequent steps this new information can be reasoned upon, or stored in a knowledge base, but the key requirement here is to be able to specify how to construct this new data with the query language. In some cases this type of new information can be modeled as an event, built from lower level conceptualizations.

Hence, we need *a declarative way of expressing derived information from incoming data streams*, as a built-in feature of the query language.

**Reasoning.** One of the main advantages of ontology-based data sources is the possibility of reasoning over them, exploiting the semantic relationships between the elements of a dataset. While query languages such as SPARQL do not impose reasoning in the evaluation semantics, it is possible to use different entailment regimes -other than *simple entailment*- that allow complex inferences on the resulting data. A whole new sub-area, under the *Stream Reasoning* (Della Valle et al., 2009) denomination, has appeared to address these challenges, using techniques such as incremental materialization or query rewriting.

A query language for ontology-based streaming processing may need considering *reasoning entailment rules for query evaluation*.

#### 4.1.1 Discussion

We analyzed the main features of an ontology-based streaming query language, and the rationale behind them. In the remainder of this chapter, we will present a query language designed to tackle them, although conditioned to the limitations stated in Chapter 3. We adopt a data model based on RDF, the W3C standard for semantic web data interchange, but extended to support the **streaming data model** requirements. Accordingly, we use SPARQL 1.1 as a basis for the SPARQL<sub>Stream</sub> language, leveraging on its **declarative and expressive** features and operators, and also adding extensions aiming at **supporting continuous queries** and recency-based constructs such as windows.

The proposed model and query language also allow **mixing static and streaming data** and expressing new information in terms of derivative queries using modifiers such as SPARQL CONSTRUCT. In turn, we will leave the handling of data imperfections to underlying streaming data engines, and restrict reasoning semantics to simple entailment.

## 4.2 RDF Stream Model

A key element for the SPARQL<sub>Stream</sub> language is the data model used to represent RDF streams. We present in this section the formalization of RDF triples and graphs, and then the extensions for representing stream graphs.

### 4.2.1 RDF triples and graphs

An RDF triple  $t$  is a tuple  $\langle s, p, o \rangle$  where  $s, p, o$  are respectively the subject, predicate and object, such that:

$$\langle s, p, o \rangle \in (I \cup B) \times I \times (I \cup B \cup L)$$

where  $I$ ,  $B$  and  $L$  are sets of IRIs, blank nodes and literals, respectively.

### 4.2.2 RDF Stream Graphs

The RDF semantics presented here is taken from the W3C RDF semantics<sup>1</sup>. An RDF stream  $S$  is defined as a sequence of pairs  $(T, \tau)$  where  $T$  is a triple  $\langle s, p, o \rangle$  and  $\tau$  is a timestamp in the infinite set of monotonically non-decreasing timestamps  $\mathbb{T}$ . More formally, the stream  $S$  is defined as follows:

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

Each of these pairs  $(T, \tau)$  can be called a *tagged triple*. The extensions presented here are based on the formalization of Pérez et al. (2009) and Gutierrez et al. (2007).

An RDF stream can be identified by an IRI, so that is possible to reference a particular stream of tagged triples, also called an RDF *stream graph*  $\langle S, g_S \rangle$ , where  $S$  is an RDF stream and  $g_S$  the IRI.

RDF streams can be *materialized* -i.e. tagged triples are physically stored in a repository- or *virtual*. In the latter case the RDF streams can be queried but their storage is not nec-

---

<sup>1</sup>RDF semantics:<http://www.w3.org/TR/rdf-mt/>



essarily as RDF tagged triples, but under another data model such as a relational stream or an event stream. This is a fact that is transparent for query users, who are typically unaware of the internal representation of the real data streams.

Stream graphs can be merged straightforwardly, under the assumption that all timestamps  $\tau$  in both graphs are sequenced in the set of infinite timestamps  $\mathbb{T}$ . The resulting merged graph will contain the union of the tagged triples of both stream graphs. In case timestamps are not synchronized, a merge operation produces an inconsistent stream graph.

Similar works on temporal-aware RDF extensions also provided mappings and translation methods to produce an equivalent normal RDF graph. For instance, (Rodríguez et al., 2009; Tappolet and Bernstein, 2009) provide mapping of the tagged triples model to standard RDF. It uses named graphs to provide temporal context to triples, i.e. a named graph is given time information, and all triples contained in it are subject to these constraints.

Nevertheless, we leave the definition of RDF streams in an abstract form for the moment, regardless of possible concrete implementations. In section 4.5 we provide details of the semantics of queries over these stream graphs, in terms of streaming relational algebra, even if these are *virtual*, i.e. they are not stored or materialized, but consumed from underlying non-RDF streaming data sources.

### 4.3 SPARQL<sub>Stream</sub> Syntax

SPARQL<sub>Stream</sub> was first introduced in (Calbimonte et al., 2010b), and has been inspired by previous proposals of streaming-oriented extensions of SPARQL, mainly C-SPARQL (Bambieri et al., 2010a), and also by continuous streaming query languages such as CQL or SNEEqL (Brenninkmeijer et al., 2008). It must be noted that SPARQL<sub>Stream</sub> emerged concurrently with other proposals -as seen in Section 2.3- and hence shares many features with them. SPARQL<sub>Stream</sub> is based on the concept of virtual RDF streams of triples that can be continuously queried, and whose elements can be bounded using sliding windows, as in C-SPARQL, and adopts the streaming evaluation semantics of SNEEqL, adapted to an RDF based model.

SPARQL<sub>Stream</sub> is based on SPARQL 1.1, and as such is able to cope with aggregates, which are an essential feature in streaming data processing. Furthermore it introduces time-based windows and triple-based windows, although we show that the latter may

lead to unexpected results. Additionally, SPARQL<sub>Stream</sub> provides window-to-stream operators that enable the generation of new streams. We present the syntax of the language below.

#### 4.3.1 SPARQL Syntax

The SPARQL language is a pattern matching based language. We define a subset of it in terms of triple and graph patterns, as in (Chebotko et al., 2009; Pérez et al., 2009). A triple pattern  $tp$  is a triple  $(sp, pp, op)$  such that:

$$(sp, pp, op) \in (I \cup V) \times (I \cup V) \times (I \cup V \cup L)$$

where  $V$  is the infinite set of variables.

A graph pattern  $gp$  is a combination of triple patterns and graph patterns, following the abstract syntax below:

$$\begin{aligned} gp &\rightarrow tp \mid \\ &gp \text{ AND } gp \mid \\ &gp \text{ OPT } gp \mid \\ &gp \text{ UNION } gp \mid \\ &gp \text{ FILTER } expr \end{aligned}$$

where AND, OPT, UNION and FILTER correspond to SPARQL conjunction, OPTIONAL, UNION and FILTER operators. In FILTER,  $expr$  is a boolean expression in terms of  $I \cup V \cup L$ , and including logical operators and other constructs defined in (Prud'hommeaux and Seaborne, 2008).

A SPARQL query is composed of these graph patterns and is of one of four query forms: ASK, SELECT, CONSTRUCT, DESCRIBE. The most common query form, SELECT, produces a result of variable bindings matching the graph pattern; a CONSTRUCT produces a new RDF graph with the query solutions; ASK produces a boolean value that is true if a solution exists; and DESCRIBE produces an RDF description of resources in the solution. A select query is defined as:

$$query \rightarrow \text{SELECT } v_1, \dots, v_n \text{ WHERE } gp$$

where  $v_1, \dots, v_n$  is a list of variables, subset of the variables of the graph pattern  $gp$ . This is the basic syntax of SPARQL. There are other constructs such as *solution modifiers* (e.g.

DISTINCT, ORDER BY, LIMIT, etc.) that are applied after pattern matching. These and other modifiers can be found in the SPARQL Query Language specification<sup>1</sup>.

A salient feature of SPARQL 1.1 is the inclusion of aggregate expressions that compute functions over groups of solutions. These are a common requirement in streaming data processing, where summarization of highly dynamic data helps providing a better understanding of recurring situations. This is achieved by optionally adding a GROUP BY solution modifier followed by the list of variables over which the solutions will be grouped, and by allowing aggregate functions in the projection list:

$$query \rightarrow \text{SELECT } f^{AGG}(v_1)|v_1, \dots \text{ WHERE } gp[\text{GROUP BY } groupvars]$$

Notice that the selection variables alternatively allow applying functions of the type  $f^{AGG}$ , such as MAX, MIN, AVG or SUM. The *groupvars* list of variables is a subset of the variables of *gp*, which is used to group the solutions. For example, the following SPARQL 1.1 query returns the maximum air temperature value reported by each sensor.

---

```

SELECT (MAX(?temperature) AS ?maxtemp) ?sensor
WHERE {
  ?obs ssn:observedBy ?sensor.
  ?obs ssn:observationResult ?res.
  ?res aemet:hasAirTemperatureValue ?val.
  ?val qu:numericValue ?temperature
}
GROUP BY ?sensor

```

---

**Listing 4.1:** Example of a SPARQL 1.1 aggregate query

Notice that the SPARQL 1.1 complete syntax includes renaming and other minor syntactic features that are described with full details in the specification.

### 4.3.2 SPARQL<sub>Stream</sub> Extensions

The SPARQL<sub>Stream</sub> syntax follows closely that of SPARQL 1.1, adding window constructs for RDF stream graphs and additional solution modifiers. In SPARQL<sub>Stream</sub> each virtual RDF stream graph is identified by an IRI, so that it can be used or referenced elsewhere in the query. To denote that a particular stream graph is used in the query, its IRI has to be referenced in the FROM clause, preceded by the NAMED STREAM keywords:

---

```

FROM NAMED STREAM <Stream_IRI>

```

---

<sup>1</sup>SPARQL 1.1 <http://www.w3.org/TR/sparql11-query/>

Notice that it is possible to add more stream graphs in the same query by adding more `FROM` clauses. Then the query dataset is the union of those graphs.

Time window definitions of the form `[start TO end SLIDE slide]` can be applied to stream graphs by appending them after the stream IRI: The *start* and *end* are expressions that indicate the lower and upper bounds of the time window, usually expressed in terms relative to the current time (expressed as `NOW`).

For instance the expression `NOW - 2 MINUTES` denotes the latest 2 minutes. The window `[NOW - 2 MINUTES TO NOW - 1 MINUTES ]` denotes an interval elapsed in the last 2 minutes, but excluding the last minute. Because streaming queries are usually interested in the latest values, the upper bound of the window is commonly the current time, in which case the `TO end` part of the window can be omitted altogether. The interval expressions are expressed relative to the current time because these queries are meant to be run continuously, hence a fixed interval is less meaningful.

Finally the optional `SLIDE` specifies how often the window will be evaluated (e.g. every minute). Note, if the size of the slide is smaller than the range of the window, then the windows will overlap in successive evaluations, if it coincides with the size of the window then every triple will appear in one and only one window, and if the slide is larger than the range, then the windows *sample* the stream.

For instance the following time window includes the triples in the last minute of the `http://example.com/mystream` stream graph, and the window is computed every minute. Notice that the expressions of the window include time units: `YEARS`, `MONTHS`, `DAYS`, `HOURS`, `MINUTES`, `SECONDS`, `MILLISECONDS`.

---

```
FROM NAMED STREAM <http://example.com/mystream> [NOW-1 MINUTES SLIDE 1 MINUTES]
```

---

As an example, the query in Listing 4.2 requests the maximum temperature and the sensor that reported it in the last hour, from the `http://aemet.linkeddata.es/observations.srdf` stream graph.

---

```
SELECT (MAX(?temp) AS ?maxtemp) ?sensor
FROM NAMED STREAM <http://aemet.linkeddata.es/observations.srdf> [NOW-1 HOURS]
WHERE {
  ?obs a ssn:Observation;
    ssn:observationResult ?result;
    ssn:observedProperty cf-property:air_temperature;
    ssn:observedBy ?sensor.
  ?result ssn:hasValue ?obsValue.
  ?obsValue qu:numericalValue ?temp.
```

```
}
GROUP BY ?sensor
```

---

**Listing 4.2:** SPARQL<sub>Stream</sub> query the maximum temperature and the sensor that measured it in the last hour

Note that it is possible to use SPARQL 1.1 features such as aggregates along with windows. The triple graph patterns and other operators are applied to the bag of triples delimited by the time window, and the query can be continuously executed, so that each time it will produce a bag of results, in this case, variable bindings. As more tuples are streamed, this will produce a continuous *stream of windows* or stream of sequences of results.

To produce the inverse result of windows, SPARQL<sub>Stream</sub> also allows generating streams of triples from windows, with what are called window-to-stream operators. These operators, namely RSTREAM, ISTREAM and DSTREAM produce a continuous stream that can be later reused or queried. They are applied after the SELECT or CONSTRUCT clauses, and their output is respectively a stream of variable bindings or triples. The difference between the three operators resides in the data that is output. With ISTREAM, only the data that was not in the previous window is added to the output stream. With DSTREAM, only the data that was in the previous windows but not in the new one is streamed. With RSTREAM all data is added to the stream.

For example, the query in Listing 4.3 generates a stream with the sensors where the temperature was previously higher than 40, but not anymore.

---

```
SELECT DSTREAM ?sensor
FROM NAMED STREAM <http://aemet.fi.upm.es/observations.srdf> [NOW-1 HOURS]
WHERE {
  ?obs a ssn:Observation;
    ssn:observationResult ?result;
    ssn:observedProperty cf-property:air_temperature;
    ssn:observedBy ?sensor.
  ?result ssn:hasValue ?obsValue.
  ?obsValue qu:numericalValue ?temp.
  FILTER (?temp > 40)
}
```

---

**Listing 4.3:** SPARQL<sub>Stream</sub> query that generates a stream of sensor identifiers where the temperature recorded has been previously higher than 40 but not anymore

### 4.3.3 Comparison of SPARQL<sub>Stream</sub> with other Languages

Other extensions for SPARQL considering streaming and time extensions have appeared in the literature as detailed in Chapter 2. Some of these proposals have been developed concurrently to SPARQL<sub>Stream</sub>. Now we briefly compare their features and main characteristics. Table 4.1 provides a summary of this comparison. While  $\tau$ SPARQL and

Language	Model	Union, Join, Optional, Filter	Aggregate	Time Windows	Triple Windows	Window-to- Stream	Sequence/ Co-occurrence
TA-SPARQL	TA-RDF	Yes	Limited (implicit grouping)	No (filters)	No	No	No
$\tau$ SPARQL	$\tau$ RDF	Yes	No	No (temporal wildcards)	No	No	No
Streaming SPARQL	RDF Streams	Yes	No	Yes	Yes	No	No
C-SPARQL	RDF Streams	Yes	Yes	Yes	Yes	No	No
CQELS	RDF Streams	Yes	Yes	Yes	Yes	No	No
SPARQL <sub>Stream</sub>	(Virtual) RDF Streams	Yes	Yes	Yes	No	Yes	No
EP-SPARQL	RDF Streams	Yes	Yes	No	No	No	Yes

**Table 4.1:** Comparative table of SPARQL<sub>Stream</sub> and other SPARQL extensions for streams.

TA-SPARQL introduced time-aware operators as extensions for SPARQL, these were oriented towards static and stored RDF, and not for streaming and continuous queries.  $\tau$ SPARQL provides temporal wildcards to denote time relations, for example intervals where a triple pattern is valid, or time relationships between intervals. TA-SPARQL provides an alternative syntax to time-based filters over a named graph that contains the temporal context. However, none of these approaches is designed for streaming query processing.

In terms of operators, Streaming SPARQL is limited in its lack of aggregates, compared to C-SPARQL, CQELS and SPARQL<sub>Stream</sub>. These three have also converged towards a SPARQL 1.1-based specification, including aggregates, property paths, among others. SPARQL<sub>Stream</sub> also adds window-to-stream operators, non-existing in other languages. It notably lacks triple based windows, mainly because they may incur in misleading results. A window in terms of triples (e.g. the latest 5 triples) is not as useful as a window in terms of an event or higher level abstraction (e.g. the latest 5 observations). Finally, EP-SPARQL is different from the other proposals as it includes sequence and co-occurrence pattern operators, typical of CEP. In our work we do not consider these operators, although we can foresee their addition as an extension for our approach.

## 4.4 SPARQL<sub>Stream</sub> Semantics

Having defined the syntax of the SPARQL<sub>Stream</sub> extensions, we now define their semantics, but for completeness we first provide a summary of the SPARQL semantics.

### 4.4.1 SPARQL Semantics

A mapping-based semantics of SPARQL has been proposed in (Pérez et al., 2009), which we present here. Given a SPARQL query  $q$  over an RDF graph, a query solution can be represented as a set  $\Omega$  of mappings  $m$ , each of which assigns terms of RDF triples in the graph, to variables of  $q$ . A mapping  $m : V \rightarrow I \cup B \cup L$ , has a domain  $dom(m)$  which is the subset of  $V$  variables over which it is defined.

The evaluation of a graph pattern  $gp$  over a dataset  $D$ , denoted as  $[[gp]]_D$  is defined as:

$$\begin{aligned} [[tp]]_D &= \{m \mid dom(m) = var(tp) \wedge m(tp) \in D\} \\ [[gp_1 \text{ AND } gp_2]]_D &= [[gp_1]]_D \bowtie [[gp_2]]_D \\ [[gp_1 \text{ OPT } gp_2]]_D &= [[gp_1]]_D \bowtie [[gp_2]]_D \\ [[gp_1 \text{ UNION } gp_2]]_D &= [[gp_1]]_D \cup [[gp_2]]_D \\ [[gp \text{ FILTER } expr]]_D &= \{m \mid m \in [[gp]]_D \wedge m \text{ satisfies } expr\} \end{aligned}$$

where  $tp$  is a triple pattern and  $var(tp)$  is the set of variables in  $tp$ , and  $gp_1, gp_2$  are graph patterns. The operations between sets of mappings ( $\bowtie, \bowtie, \cup, \setminus$ ) are given by:

$$\begin{aligned} \Omega_1 \bowtie \Omega_2 &= \{m_1 \cup m_2 \mid m_1 \in \Omega_1, m_2 \in \Omega_2 \text{ are compatible}\} \\ \Omega_1 \cup \Omega_2 &= \{m \mid m \in \Omega_1 \vee m \in \Omega_2\} \\ \Omega_1 \setminus \Omega_2 &= \{m \in \Omega_1 \mid \forall m' \in \Omega_2, m \wedge m' \text{ are not compatible}\} \\ \Omega_1 \bowtie \Omega_2 &= (\Omega_1 \bowtie \Omega_2) \cup (\Omega_1 \setminus \Omega_2) \end{aligned}$$

Compatibility of mappings is defined as follows: mappings  $m_1$  and  $m_2$  are compatible if  $\forall x \in dom(m_1) \cap dom(m_2)$ , then  $m_1(x) = m_2(x)$ .

This semantics is based on the set of operators identified in SPARQL 1.0. The SPARQL 1.1 specification allows additional constructs, including aggregates as we saw in Section 4.3.1. The evaluation of group-by and aggregate functions in a graph pattern  $gp$  is given be-

low<sup>1</sup>.

$$\begin{aligned} [[gp \text{ GROUP BY } exprlist]]_D &= Groupby(exprlist, [[gp]]_D) \\ [[AGG_f agglstgp]]_D &= Agg_f([[gp]]_D, agglst) \end{aligned}$$

where  $exprlist$  is a list of expressions  $\{expr_1, \dots, expr_n\}$  by which the solutions are grouped.  $agglst$  is the list of expressions to which the aggregation function  $f$  is applied. The semantics of the  $Groupby$  and  $Agg_f$  are given below.

$$\begin{aligned} Groupby(exprlist, \Omega) &= \{ListEval(exprlist, m) \rightarrow \\ &\quad \{m' \mid m' \in \Omega, ListEval(exprlist, m) = ListEval(exprlist, m')\} \mid m \in \Omega\} \\ Agg_f(keyvals, agglst) &= \{(key, F(\Omega)) \mid key \rightarrow \Omega \in keyvals\} \end{aligned}$$

where  $ListEval$  is a function that evaluates a list of expressions over a mapping:

$$ListEval(exprlist, m) = \{e_i \mid \forall expr_i \in exprlist, e_i = expr_i(m)\}$$

$expr(m)$  is the value resulting from the expression  $expr$ , replacing the variables by the terms given by  $m$ .  $keyvals$  is a set of partial functions from keys to solutions:  $\{key_1 \rightarrow \Omega_1, \dots, key_n \rightarrow \Omega_n\}$ , and  $F(\Omega)$  applies the aggregation function  $f$  of  $Agg_f$  to the groups of solutions as follows:

$$F(\Omega) = f(\{ListEval(agglst, m) \mid m \in \Omega\})$$

The aggregation function  $f$  is one of MAX, MIN, AVG, SUM, COUNT or other similar function that reduces a list of values (following the SPARQL 1.1 semantics).

#### 4.4.2 SPARQL<sub>Stream</sub> Extensions Semantics

The extensions provided by SPARQL<sub>Stream</sub> are formalized below, as in (Calbimonte et al., 2010b, 2012a). We define a stream of windows as a sequence of pairs  $(\omega, \tau)$  where  $\omega$  is a set of triples, each of the form  $\langle s, p, o \rangle$ , and  $\tau$  is a timestamp in the infinite set of timestamps  $\mathbb{T}$ , and represents when the window was evaluated. More formally, we define the triples that are contained in a time-based window evaluated at time  $\tau \in \mathbb{T}$ ,

---

<sup>1</sup>We follow the semantics described in the SPARQL 1.1 specification: <http://www.w3.org/TR/sparql11-query>



denoted  $\omega^\tau$ , as

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

where  $t_s, t_e$  define the start and end of the window time range respectively, and may be defined relative to the evaluation time  $\tau$ . Note that the rate at which windows get evaluated is controlled by the SLIDE defined in the query, which is denoted by  $\delta$ , affecting only the concrete values of  $t_s, t_e$ .

We define the three window-to-stream operators as

$$\begin{aligned} \text{RSTREAM}((\omega^\tau, \tau)) &= \{(\langle s, p, o \rangle, \tau) \mid \langle s, p, o \rangle \in \omega^\tau\} \\ \text{ISTREAM}((\omega^\tau, \tau), (\omega^{\tau-\delta}, \tau - \delta)) &= \{(\langle s, p, o \rangle, \tau) \mid \langle s, p, o \rangle \in \omega^\tau, \langle s, p, o \rangle \notin \omega^{\tau-\delta}\} \\ \text{DSTREAM}((\omega^\tau, \tau), (\omega^{\tau-\delta}, \tau - \delta)) &= \{(\langle s, p, o \rangle, \tau) \mid \langle s, p, o \rangle \notin \omega^\tau, \langle s, p, o \rangle \in \omega^{\tau-\delta}\} \end{aligned}$$

where  $\delta$  is the time interval between window evaluations. The  $\omega^{\tau-\delta}$  represents the window immediately before  $\omega^\tau$ . Note that RSTREAM does not depend on the previous window evaluation, whereas both ISTREAM and DSTREAM depend on the contents of the previous window.

#### 4.4.3 Comparison of SPARQL<sub>Stream</sub> Semantics with other Languages

SPARQL<sub>Stream</sub> semantics are based on SPARQL 1.1, and the rationale behind its formal specification is that windows lead to bounded triples that fit into standard SPARQL operators, following the mapping-based semantics of Pérez et al. (2009). This idea draws from CQL and also the formal definition of subsequent DSMS languages such as SNEEql. It was first formalized in this way in the C-SPARQL semantics (Barbieri et al., 2010b), although the aggregations semantics later changed towards SPARQL 1.1, as already noted in Chapter 2. In addition, C-SPARQL defines the semantics for triple-based windows  $\omega_p$  (called *physical*):

$$\omega_p(R, n) = \{(\langle s, p, o \rangle, \tau) \in \omega_l(R, t_s, t_e) \mid c(R, t_s, t_e) = n\}$$

Where  $\omega_l$  is a time window,  $c$  is the cardinality of the triples of  $R$  in a time interval. CQELS semantics follow a similar definition, although leaves the window specification open to different modes. These models are different to Streaming SPARQL, which redefines the SPARQL semantics altogether to allow window definitions in any triple pattern,

instead of the RDF stream graphs windows of the other approaches. EP-SPARQL additional operators of sequence and temporal co-occurrence require additions to the query semantics, and can be found in (Anicic et al., 2011).

## 4.5 Rewriting SPARQL<sub>Stream</sub> Queries

In the previous sections we have provided the syntax and semantics of SPARQL<sub>Stream</sub>. These are abstract formalizations independent of how the streaming data is internally managed, e.g. as materialized native RDF streams or as virtual RDF streams. In this thesis, we are focused on the case where streaming data is already managed by DSMS, CEP or streaming data middleware technologies. In this section we provide the operational semantics of SPARQL<sub>Stream</sub> through query rewriting, when the data source is not an RDF stream but a sensor network-based or an event-based stream, and given mappings that relate one model to the other. In this case we need to transform the SPARQL<sub>Stream</sub> queries into requests that can be processed by the corresponding heterogeneous streaming data sources.

In this section we formalize the semantics of SPARQL<sub>Stream</sub> in terms of algebra expressions of relational streams, given mapping definitions that relate ontology models to relational stream schemas. The main goal of this formalization is to provide theoretical foundations for transforming a set of queries over an ontological schema, into expressions that can be serialized as streaming query languages used to access the data sources. We analyze the query rewriting based on abstract mapping definitions that later can be implemented in languages such as R2RML as we will see in Chapter 5. This work is based partially on the formalization of (Calvanese et al., 2005), and extends those of (Calbimonte et al., 2010b, 2012a).

### 4.5.1 Relational to RDF Mappings

The definition of mappings we use, is based on relational-to-RDF mappings. A relation  $r$  is defined by a relation schema with attributes  $\vec{a}$ . Each attribute has a name and a datatype from a set of basic datatypes. For instance the relation schema  $Sensor_{name,description}$  represents a sensor with a *name* and *description* attributes.

RDF triples are instead represented as subject-predicate-object tuples, as detailed in Section 4.2. Mappings from a relation schema  $r$  to RDF triples can be defined as functions of the form  $\mu(r_{\vec{a}}) = \{(s, p, o) \mid s \in I \cup B, p \in I, o \in I \cup B \cup L\}$ :

The function  $\mu$  generates  $s, p, o$ , the triple subject, predicate and object of the triple respectively.

#### 4.5.2 Definition of Mappings for Query Rewriting

Query rewriting requires mapping assertions that describe how to construct triple instances in terms of an ontology model, from a given relational stream schema, as we saw in the previous section. We use the same abstraction for a stream  $s$ , whose schema can be defined analogously, as a set of attributes  $\vec{a}$ . Then, a mapping assertion  $\mu$  is defined as a function over a stream  $s$  :

$$\mu(s_{\vec{a}}) \rightarrow (f_{\mu}^s(s_{\vec{a}}), f_{\mu}^p(s_{\vec{a}}), f_{\mu}^o(s_{\vec{a}}))$$

where  $s$  is a logical stream with attributes  $\vec{a} = \{a_1, a_2, \dots, a_m\}$ , each of them with a certain data type. Notice that this logical stream can be a view over several real streams (e.g. as virtual GSN streams), however in the rest of this section we will not discuss this case and will just assume that the logical stream has a number of attributes, and one of them is the timestamp. The mapping  $\mu$  explains how to construct a triple, with the subject IRI or blank node generated by  $f_{\mu}^s$ , the predicate IRI generated by  $f_{\mu}^p$  and the object IRI, blank node or literal generated by  $f_{\mu}^o$ .

These functions can generate IRIs, blank nodes or literals in different ways, using the attributes  $\vec{a}$ , for instance by concatenating the values of the attributes, by simply copying the attribute value as is, or even using a constant value (e.g. a constant predicate IRI is a common function for  $f_{\mu}^p$ ). Notice that these functions, if ill defined, may violate the RDF model (e.g. a subject function generating a literal value), in which case the generated triple is ignored<sup>1</sup>.

As an example, consider a stream  $s1$  with attributes  $ts, speed$  representing the timestamp and wind speed values respectively. A simple mapping that generates an observation of type `aemet:WindSpeedObservation`<sup>2</sup> for each tuple of  $s1$  is given below:

$$\mu_1(s1_{ts, speed}) \rightarrow (f_{\mu_1}^s(s1_{ts, speed}), \text{rdf:type, aemet:WindSpeedObservation})$$

Notice that  $f_{\mu_1}^o$  and  $f_{\mu_1}^p$  are constant functions and are directly represented using the

---

<sup>1</sup>We assume the conventions adopted by the W3C RDB2RDF Working Group.

<sup>2</sup>For simplicity we use RDF URI prefixes in the examples, e.g. `aemet` denotes <http://aemete.linkeddata.es/ontology#>

corresponding constant value (`rdf:type` and `aemet:WindSpeedObservation`). The functions for the generation of URI values can be simply a concatenation of strings and values of attributes of  $s$ . For instance the function  $f_{\mu_1}^s(s1_{ts, speed})$  can be defined as prefixing the  $ts$  attribute with a constant string:

$$f_1^s(s1_{ts, speed}) = \text{CONCAT}('http://aemet.linkeddata.es/observations/windspeed/s1', s1.ts)$$

Alternatively, the object of the triple can be the subject generation function  $f_{\mu'}^s$  of another mapping  $\mu'$ :

$$\mu(s_{\vec{a}}) \rightarrow (f_{\mu}^s(s_{\vec{a}}), f_{\mu}^p(s_{\vec{a}}), f_{\mu'}^s(s_{\vec{a}} \bowtie_{cond} s'_{\vec{a}}))$$

In this case the function for generating the triple object uses a *parent* triple mapping  $\mu'$  with another stream  $s'$  and attributes  $\vec{a}'$ , applied to the combined tuples of  $s, s'$  that match the condition  $cond$ .

As an example, consider the following additional mapping assertions, that generate more triples from each tuple of  $s1$ :

$$\begin{aligned} \mu_2(s1_{ts, speed}) &\rightarrow (f_{\mu_1}^s(s1), \text{ssn:observationResult}, f_{\mu_3}^s(s1)) \\ \mu_3(s1_{ts, speed}) &\rightarrow (f_{\mu_3}^s(s1), \text{rdf:type}, \text{aemet:WindSpeedResult}) \end{aligned}$$

In  $\mu_3$  we generate a new triple of type `aemet:WindSpeedResult` with a different subject defined by  $f_{\mu_3}^s$ . The triple generated in  $\mu_2$  has the particularity of linking the subject of the mapping  $\mu_1$ , with the subject of  $\mu_3$ , through the constant predicate `ssn:observationResult`. In this particular case we omitted the join condition as the stream is always  $s1$  for all mappings, but it can be represented as  $s1 \bowtie_{s1.ts=s1.ts} s1$  if we take  $ts$  as a tuple identifier.

We can add more mapping assertions to this example, including more complex triple generation rules, and literal objects:

$$\begin{aligned} \mu_4(s1_{ts, speed}) &\rightarrow (f_{\mu_3}^s(s1), \text{ssn:observationValue}, f_{\mu_5}^s(s1)) \\ \mu_5(s1_{ts, speed}) &\rightarrow (f_{\mu_5}^s(s1), \text{qu:numericalValue}, f_{\mu_5}^o(s1)) \end{aligned}$$

$\mu_4$  links the subjects of  $\mu_3$  and  $\mu_5$  through a constant predicate. In  $\mu_5$  we generate a triple object using a function, defined in terms of the attributes of  $s1$ . For instance  $f_{\mu_5}^o(s1)$  can

be simply the value of *speed*:

$$f_{\mu_5}^o(s1) = s1.speed$$

Given a set of mappings  $M$  over a set of streams  $s_j$ , it is possible to follow two approaches for accessing the data of  $s_j$  in terms of the ontology mapped by  $M$ :

- (i) Materialization: generating all possible triples and exposing them through a SPARQL endpoint, or
- (ii) Virtual RDF streams: executing a query rewriting algorithm from incoming SPARQL<sub>Stream</sub> queries, to queries in terms of the streams  $s_i$ .

The first approach is straightforward but is unsuitable for continuous query processing as it requires all incoming data to be materialized at each time. The second one requires an algorithm for query rewriting, and a SPARQL<sub>Stream</sub> aware endpoint. We will discuss our approach in the following sections.

### 4.5.3 Rewriting to Algebra Expressions

Our query rewriting approach can be executed against DSMS, CEP or streaming middleware, and therefore we do not bind our approach to any particular technology. Specifically, we describe the rewriting in terms of abstract algebra expressions, following the work of (Chebotko et al., 2009).

First we define the function *findMapping* that, given a triple pattern  $tp$  and a set of mappings  $M$ , returns the mapping  $\mu$  whose stream  $s$  may contain triples that match  $tp$ .  $\mu$  is found among the mappings of  $M$ , as described in Section 4.5.2. For example, given the mappings  $M = \{\mu_1, \mu_2, \mu_3\}$  of Section 4.5.2 and the triple pattern  $tp$ :

$$tp = (?obs \text{ rdf:type } aemet:WindSpeedObservation )$$

the stream  $s1$  of mapping  $\mu_1$  may contain tuples that are able to produce triples matching  $tp$ . Therefore  $findMapping(tp, M) = \mu_1$ . Variables in  $tp$  are prefixed with '?' in this notation.

Notice that the mapping provides information about: 1) from which stream  $s$ , the triples will be generated, and 2) which attributes of the stream  $s$  are needed to produce the subject, predicate and object of the triples that match  $tp$ . This is specified by the

functions  $f^s, f^p, f^o$  of mapping  $\mu$ , as described in Section 4.5.2. We formalize this in the semantics of the evaluation of  $tp$ .

Given a triple pattern  $tp = (sp, pp, op)$ , the semantics of its evaluation over a set of relational streams referenced by a set of mappings  $M$ , is given by  $eval(tp, M)$ , which is an algebra expression defined as:

$$eval(tp, M) = \rho_{f^s \rightarrow sp, f^p \rightarrow pp, f^o \rightarrow op} \pi_{f^s, f^p, f^o}(s)$$

where  $\rho$  is the relational rename operation and  $\pi$  is the relational projection operation.  $s$  is the stream referenced by the mapping  $\mu = findMapping(tp, M)$  and  $f^s, f^p, f^o$  are the functions of  $\mu$  that generate the projection expressions for producing respectively the subject, predicate and object, for every tuple of  $s$ .

For the previous example, the evaluation of  $tp_1$  is given by:

$$eval(tp_1, M) = \rho_{f^s \rightarrow sp, f^p \rightarrow pp, f^o \rightarrow op} \pi_{f^s(s1.ts), f^p(), f^o()}(s1)$$

The resulting algebra expression projects the  $s1.ts$  attribute, applying the  $f^s$  function to create the subject. The functions  $f_{\mu_1}^p$  and  $f_{\mu_1}^o$  in this case are constants, so the predicate and object are the same for all tuples of  $s1$ . For the evaluation of more complex graph patterns, the semantics are given by more complex algebra expressions that are described below.

Given a graph pattern  $gp = gp1 \text{ AND } gp2$ , the evaluation over mappings  $M$  is defined as:

$$eval(gp, M) = eval(gp1, M) \bowtie eval(gp2, M)$$

For a graph pattern  $gp = gp1 \text{ OPT } gp2$  the evaluation over  $M$  is given by:

$$eval(gp, M) = eval(gp1, M) \bowtie eval(gp2, M)$$

For a graph pattern  $gp = gp1 \text{ UNION } gp2$  the evaluation over  $M$  is given by:

$$eval(gp, M) = eval(gp1, M) \uplus eval(gp2, M)$$

For a graph pattern  $gp = gp1 \text{ FILTER } expr$  the evaluation over  $M$  is given by:

$$eval(gp, M) = \sigma_{expr}(eval(gp1, M))$$

As an example consider the following query pattern  $gp$ :

$$\begin{aligned} gp = & (?obs \text{ rdf:type aemet:WindSpeedObservation } ) \text{ AND} \\ & (?obs \text{ ssn:observationResult ?res} ) \text{ AND} \\ & (?res \text{ ssn:observationValue ?val} ) \text{ OPT} \\ & (?val \text{ qu:numericValue ?speed} ) \end{aligned}$$

By applying the evaluation rules, we find the corresponding mapping for each triple, and then apply the rules for the AND and OPT graph patterns and we obtain the following algebra expression:

$$eval(tp, M) = \pi_{fs}(s_1) \bowtie (\pi_{fs,fo}(s_1) \bowtie (\pi_{fs,fo}(s_1) \bowtie \pi_{fs,fo}(s_1)))$$

#### 4.5.4 Rewriting Window Operators

In the previous section we presented the evaluation semantics of a graph pattern query with respect to a set of mappings, in terms of relational algebra expressions. For the additional constructs of SPARQL<sub>Stream</sub>, mainly windows and window-to-stream operators, we provide similar descriptions of the evaluation semantics.

For a triple pattern  $tp$ , in a stream graph  $G$  with a window bounded in  $\{t_s, t_e\}$  and a slide  $\delta$ :  $gp = tp \text{ WINDOW }_{t_s, t_e, \delta} G$  the evaluation over  $M$  is given by:

$$eval(gp, M) = \omega_{t_s, t_e, \delta}(eval(tp, M))$$

In this case,  $\omega_{t_s, t_e, \delta}$  is the window operator that takes only those tuples whose timestamp is in the  $t_s, t_e$  interval. In general, the  $\omega$  operator is the innermost operator in the algebra expression, being applied directly over the relational streams.

In the case of window-to-stream operators, they are applied as solution modifiers as the DISTINCT construct, and its application is straightforward.

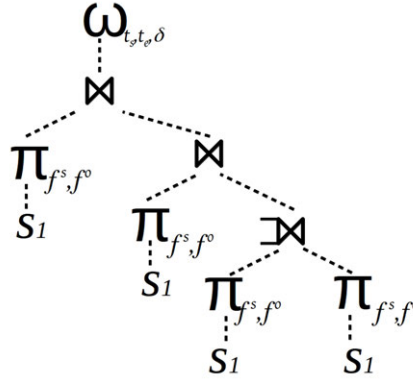
For example, if we modify the previous query pattern to include a window:

$$\begin{aligned} gp = & (?obs \text{ rdf:type aemet:WindSpeedObservation } ) \text{ AND} \\ & (?obs \text{ ssn:observationResult ?res} ) \text{ AND} \\ & (?res \text{ ssn:observationValue ?val} ) \text{ OPT} \\ & (?val \text{ qu:numericValue ?speed} ) \text{ WINDOW } G_{t_s, t_e, \delta} \end{aligned}$$

Then, the evaluation of  $gp$  can be represented as the following algebra expression:

$$eval(tp, M) = \omega_{t_s, t_e, \delta}(\pi_{f^s}(s_1) \bowtie (\pi_{f^s, f^o}(s_1) \bowtie (\pi_{f^s, f^o}(s_1) \bowtie \pi_{f^s, f^o}(s_1))))$$

This expression can be represented as a tree (Figure 4.1), where the leaf nodes are the streams and the other nodes are the relational streaming operators.



**Figure 4.1:** Tree representation of the evaluation of a SPARQL<sub>Stream</sub> query rewritten as an algebra expression.

#### 4.5.5 Rewriting other Operators

Query rewriting can be defined for other operators such as aggregates. An aggregate query pattern is rewritten as follows:

$$(Agg_f \text{ aggrlist } gp \text{ group by } exprlist, M) =_{exprlist'} G_{f(aggrlist')}(eval(gp, M))$$

Where  $G$  is a grouping aggregator relational operator. The  $exprlist'$  is the list of expressions for grouping, corresponding to the original  $exprlist$ . Rewriting is only possible if the aggregation functions  $f$  in the original query have their equivalents in the relational expression.

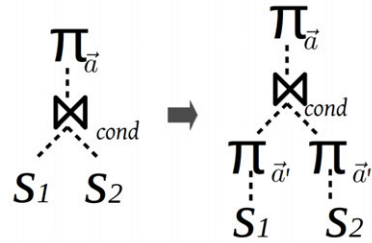
In the case of solution modifiers for window-to-stream transformation (ISTREAM, DSTREAM, RSTREAM), these only add a corresponding modifier operator on top of the algebra expression tree. Other modifiers, such as duplicate removal follow the same principle.



## 4.6 Query Optimization

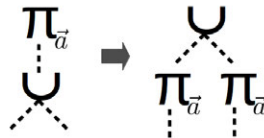
In the previous section we described the formal semantics of  $\text{SPARQL}_{\text{Stream}}$  based on query rewriting to algebra expressions. However, the resulting expressions can be inefficient at the time of execution. Therefore, it is important to consider using logical equivalences on these expressions, taken from well known relational static optimization techniques.

**Push-down projection** Expressions resulting from query rewriting have projections already very close to the stream relations, because they are produced at the time of translating a triple pattern. However, other projections, such as the one resulting from the SELECT clause in a  $\text{SPARQL}_{\text{Stream}}$  query, or the ones in the tree that resulted from various optimization steps, can be subject to the classical pushdown rule (Abiteboul et al., 1995). The intuition behind this optimization rule is that by pushing down the projection operator, we exclude earlier the attributes that are not needed for the evaluation. This can be applied in unary operators and also in binary operators such as join and union. For the join (see Figure 4.2), the projection is distributed to the join children, although the new projected attribute list is the union of the original ones  $\vec{a}$  and those present in the join condition  $\text{cond}$ .



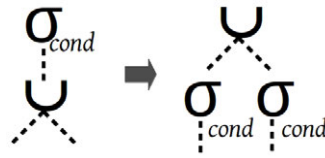
**Figure 4.2:** Push down projection.

In the case of the union, the distribution of the projection is straightforward (Figure 4.3), it is applied to all children of the union.



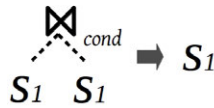
**Figure 4.3:** Push down projection in a union.

**Push-down selection** Selection operators are commonly generated by `FILTER` operators, or triple patterns with a fixed object or subject, and may appear anywhere in the algebra expression tree. Another common optimization heuristic is to push down selections, with the intuition that they will produce less intermediate results. In Figure 4.4, the selection is pushed down to a union operator, distributing the operator to the union children. In case of joins, the pushdown is not always possible, if it conflicts with the join condition.



**Figure 4.4:** Push down selection.

**Self-join simplification** While in SQL queries and similar relational query languages, self joins are rare, or the result of a bad query/database design, in query rewriting they can appear often. For instance, when two or more triple patterns match a mapping with the same logical stream, they will produce a join on that stream. These self-joins, if detected, can simplify the resulting algebra expression, eliminating the join altogether (Figure 4.5). However, the self-join can only be eliminated if the join condition is an equality on a unique key.



**Figure 4.5:** Self-join simplification.

**Empty-join simplification** After the query optimization process, it is possible that some branches of the algebra tree are simplified (e.g. after a self-join elimination), resulting in empty relations on a leaf node. If one of the children of a join is an empty relation, then the whole join is empty as well, as illustrated in Figure 4.6. This simple optimization step can quickly eliminate the need for sending unnecessary joins to the underlying query processor.

**Push-down window** The window operator  $\omega$  can drastically reduce the number of intermediate tuples to be processed, as it limits the evaluation to a subset defined by

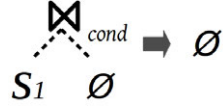


Figure 4.6: Empty join simplification.

the window boundaries. We push down the window operator as near as possible to the leafs, in order to cut down the number of tuples at the earliest stage of the evaluation, as depicted in Figure 4.7. This optimization step is possible for time-based windows, which are specified at the stream graph level. Therefore it is necessary that the mappings specify to which stream graph the generated triples will belong.

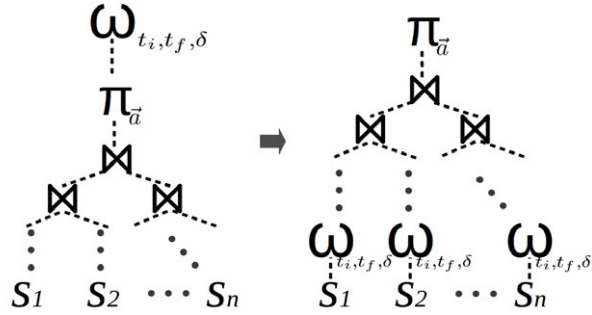


Figure 4.7: Push down window.

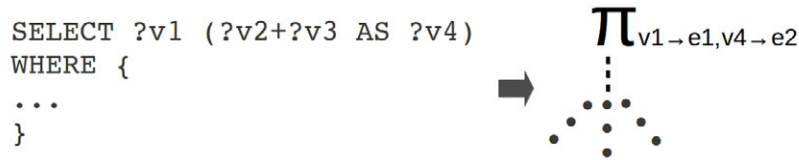
**Empty selection** A special case that is quite common in query rewriting is the presence of constants in the triple mappings. In these cases the subject or object is a projection of a constant function, that can be used in query optimization. If there is a selection filter whose condition cannot possibly match the constant value, the resulting relation is empty and it can further simplify the algebra tree.

## 4.7 Query Results Translation

In Section 4.5 we proposed a query rewriting formalization of the semantics of the evaluation of SPARQL<sub>Stream</sub>. The results of the rewritten queries, which are bags of tuples, need to be transformed into results that conform to the SPARQL<sub>Stream</sub> query form: SELECT, CONSTRUCT or ASK. We show that in fact, for the SELECT query type the results do not need any transformation, and for the other two, the modifications are minimal.

### 4.7.1 Projected Expressions

In a `SELECT` query, the rewritten algebra expression is expected to project the expressions that constitute the answer of the query. An expression can be simply a variable, or even an aggregate or a function. Each of these expressions is renamed to a variable name that matches a corresponding `SPARQLStream` variable. `SELECT` queries return results as variable bindings ( $var \rightarrow value$ ), where  $var$  is the variable name and  $value$  is the data value retrieved from the expression evaluation. For example, in Figure 4.8, the projected variables in the `SPARQLStream` query are  $v1$  and  $v4$  (which is a complex projection expression that depends on other variables, namely  $v2$  and  $v3$ ). These two variables are exposed by the rewritten algebra expression using a renaming operation (e.g.  $v1 \rightarrow e1$ ) over the expressions  $e1$  and  $e2$ , which represent the computed values in the algebra expression.



**Figure 4.8:** Projected expressions in query results

### 4.7.2 Construct and Ask

In the case of a `CONSTRUCT` query, the projection of the algebra expression provides the variable bindings of the construction template. Then, the only transformation needed consists in replacing the projected variables in the construction template, instead of producing variable bindings. This is an implementation step that does not affect the generation of the algebra expression.

Finally, for the `ASK` query, the absence of a solution will throw a `false` value as a result, and otherwise the boolean `true` value. In some query languages a conditional operation can be set up to simulate this behavior, or otherwise it can be implemented out of the algebra expression by checking the result size.

## 4.8 Discussion

In this chapter we presented the `SPARQLStream` language, its syntax and semantics. We also presented its operational semantics in terms of relational stream algebra. This

contribution provides a formal specification of how SPARQL<sub>Stream</sub> queries can be rewritten into languages based on relational models. We will explore in the next chapter the query rewriting process in detail, and see that DSMS, CEP and sensor middleware can be queried using particular instantiations of the stream relational algebra.

Moreover, we detailed how the query rewriting uses mapping definitions that describe how a triple in a virtual stream graph, can be produced from tuples of a relational stream. The formalization proposed in this chapter with respect to mappings, can be implemented using standard mapping languages, such as R2RML, as we will see in the next chapter.

The mapping-based rewriting described in this thesis, differs from other existing RDF SPARQL streaming approaches, in that it relies on existing stream query processing engines, while allowing high flexibility of mappings from relational streams to RDF streams. This flexibility is a key asset that allows using SPARQL<sub>Stream</sub> combined with a wide range of query processing technologies, as detailed in the next chapter, going beyond existing approaches that are either attached to a predefined schema and engine, or others that need to re-implement a query evaluator from scratch.

Finally, we presented static optimization techniques, taken from relational algebra, that can be applied to SPARQL<sub>Stream</sub> rewritten queries. These optimizations may not be necessary in an implementation that already performs them internally, but because SPARQL<sub>Stream</sub> query engines range from advanced DSMS to very simple sensor middleware, we stress their importance. A low-expressiveness language of a sensor middleware seldom includes advanced operators such as joins and unions, so these optimizations may allow a query to be executable at all. We did not discuss in detail other possible dynamic optimizations, based on heuristics, cost-based models and adaptivity, that could be useful in more complex scenarios, when part of the query evaluation is performed by the underlying query processor and the other is performed by the SPARQL<sub>Stream</sub> execution engine.

## Chapter 5

# Querying Semantic Sensor Networks

Making use of streaming data, produced by sensor networks and other dynamic data services, is a challenging task, considering its intrinsic heterogeneity. Data models and schemas from different streaming data sources may be different, the data types and structures are not always compatible, and even the data values often use different representations. Furthermore, even with agreements on the data types and structures, higher-level models and concepts can be useful to represent events and situations derived from raw streaming data.

Ontologies have been successfully used as conceptual models that represent a certain domain. They provide explicit semantic information that can be exploited by data consumers, and they can be used to describe queries declaratively. In this chapter we explore techniques that allow querying raw streaming data from different types of streaming data sources, exposed in terms of high-level ontological concepts and terms, through a `SPARQLStream` query façade with streaming capabilities.

First, we show how ontologies can be used to represent streaming data, such as sensor observations and the context of their measurements, and how we can describe mappings that relate these ontological concepts to raw streams. Then, using these mappings, we propose techniques that allow ontological queries - e.g in `SPARQLStream` - to be rewritten as queries in terms of the raw streams. These techniques rely on the theoretical foundations of the language semantics and query transformation presented in [Chapter 4](#).

## 5.1 Design of an Ontology-based Stream Query Processor

In this section we provide the design principles of an ontology-based streaming data processor that exposes the data in terms of rich ontological models, using query rewriting. We provide details that span from the specification of mappings from relational to ontological models, to query processing and evaluation.

### 5.1.1 Ontologies for describing Streaming Data

In order to pose queries to streaming data sources in terms of ontologies, it is first necessary to define the vocabularies describing these models. This modeling is usually largely specific to the domain and use-case, but there are common elements that can be captured in high level ontologies. Ontology modeling falls outside of the scope of this work, but we can summarize the elements that are generally included in streaming data queries:

- **Time-related concepts:** these usually include the time when an observation was made, or when it was recorded. Although these are usually specific points in time, they can be intervals as well.
- **Location:** geographical coordinates, global, or relative to a reference point, indicating where an action, event or observation takes place.
- **Observer:** the entity, device or person that produced or authored the observation data.
- **Feature:** the entity or phenomena or subject of the observation data.
- **Property:** related to the description of the type of observation or specific property that is measured or observed.
- **Provenance:** describes the origin or sequence of events, actions or sources of the data.
- **Quality:** indicates the quality of the streaming data, which can be related to trust, uncertainty, noise, etc.

While some of these elements may be supported to a limited degree in some streaming queries and data models, they correspond to key requirements and are part of the current challenges in streaming semantic modeling.

### 5.1.2 Declarative Mappings from Streams to Ontologies

Rich ontology models, require to be mapped to the low-level schemas of streaming data sources, in order to be queried. These mappings may use different approaches such as GAV, LAV or GLAV to represent the correspondences between the different models. They also may require dealing with different types of native streaming schemas, because these may vary depending on the underlying streaming or event processing system.

Declarative mapping specify what are the equivalences between model schemas, and describe them in implementation agnostic ways. We presented a formal specification of such type of mappings in Section 4.5.2, but it would need to be written as a concrete mapping language. Moreover, it is desirable that a mapping language follows a standard, so as to allow reuse, wider adoption and avoid redefinition of languages and steep learning curves. We propose using R2RML, the W3C Recommendation for RDB2RDF mapping language (seen in Section 2.2.2) for specifying this type of mappings, even if the underlying data sources are not relational databases but DSMS, CEP or sensor middleware.

### 5.1.3 Querying in terms of Ontologies

We presented in Section 4.1 the requirements of ontology-based streaming query languages, which are suited for this task. In Chapter 4 we presented SPARQL<sub>Stream</sub>, an example of such language, which follows these guidelines:

- Extending SPARQL with window operators in stream graphs.
- Extending SPARQL with window-to-stream operators.
- Allowing combining static and dynamic triple and graph patterns.
- Allowing aggregation and grouping operators.
- Allowing construction of streams of new instances.
- Querying RDF streams continuously

These features allow querying data streams as if they were instances of high-level ontologies, that represent Events, Observations, including temporal, spatial and contextual concepts.

### 5.1.4 Rewriting queries to native streaming languages

In the case of query-rewriting to underlying native query languages, such as the case of SPARQL<sub>Stream</sub>, it is required that clear semantics are provided, so as to fully understand



how the original queries are transformed into requests that the streaming or event processor can evaluate.

However, the target query language expressiveness may be different, depending on the features available in the underlying system. For instance, a sensor middleware may not expose the same rich operators available in a fully blown complex event processor. Therefore it is required that the evaluation semantics are formally specified in an abstract way, and that this formalization can be instantiated in a target language, converting the abstract operator in syntactically correct queries or requests. In case that the query cannot be rewritten, the rewriting is not possible and an error explaining the causes is expected. It is possible that in some cases, operations unavailable in the target language can be implemented out of the query processor as post-operations. Simple unions, formatting and simple functions are candidates for these operations.

### 5.1.5 Delegation of Query Execution

Depending on the type of query processing engine: DSMS, CEP or streaming middleware, and the query capabilities available, the delegation of query execution may differ greatly. Given the lack of wide-spread standards in querying streams and events, the query interfaces are normally different in terms of query language, access and protocol. Given this heterogeneity, it is required that ontology-based query processing can delegate execution to different systems in a pluggable way, by means of query adapters and execution adapters.

These adapters, are not required to transform all the data but are expected to handle the system-specific details of query languages and query access and protocols, following a mediation adapter model ([Wiederhold and Genesereth, 1997](#)). These include the ability to:

- Set up a stream
- Pose a snapshot query to streams
- Pose a continuous query to streams
- Retrieve snapshot query results
- Retrieve query results continuously
- Subscribe to query results

All these operations should be executed in the adapter and must not be exposed directly to the client. The plug-in nature of adapters should allow adding support to new type of systems in a transparent way to the querying clients.

### 5.1.6 Query Results Delivery

As a final requirement, the delivery of results should be provided in different ways, depending on the type of query interaction. These may include:

- Delivering instant results to a snapshot SELECT, CONSTRUCT or ASK query.
- Delivering continuous results to clients, at request or pull-mode.
- Delivering requests to the client in push-mode through notifications.
- Results delivery as bindings or RDF depending on the type of query, and in different formats depending on the request.

## 5.2 Mapping Streams to Ontologies

Our proposal for ontology-based data access to streaming data relies on the use of ontologies, to model the incoming information, and mappings that relate the raw stream schemas to those shared ontologies. Streaming sources such as sensor networks produce raw and often unstructured data streams, as the example in Listing 5.5. It shows two sensors (*wan7* and *imis\_wfbe*) that have different schemas, although they both measure *wind speed*.

---

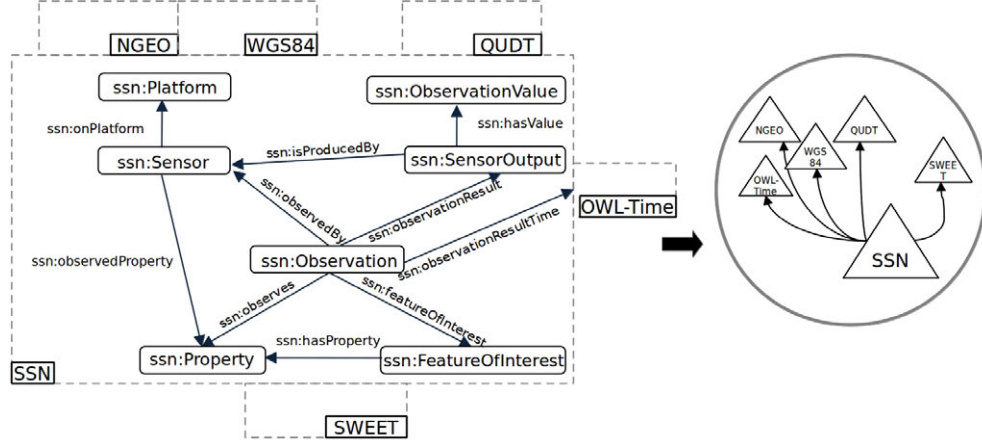
```
wan7:      {wind_speed_scalar_av FLOAT,timed DATETIME}  
imis_wfbe: {vw FLOAT,timed DATETIME}
```

---

**Listing 5.1:** Heterogeneous sensor stream schemas

There is no commonly-agreed metadata that indicates that these sensors measure wind speed. Consequently if we want to query these observations, we need to know the names of the sensors and the names of all the attributes that correspond to the semantic concept of wind speed. This error-prone task simply turns unfeasible when the number of available sensors is large.

Using an ontology network such as the one in Figure 5.1, based on the SSN ontology seen in Section 2.4.2, we can represent sensor data measurements in terms of higher level concepts. For instance consider the sensor observation in Listing 5.2.



**Figure 5.1:** The SSN Ontology combined with domain ontologies, geo and temporal vocabularies forming an ontology network.

```

swissex:WindSpeedObservation1
  rdf:type ssn:Observation;
  ssn:featureOfInterest [rdf:type sweetAtmoWind:Wind];
  ssn:observedProperty [rdf:type sweetSpeed:WindSpeed];
  ssn:observationResult [rdf:type ssn:SensorOutput;
  ssn:hasValue [qudt:numericValue "6.245"^^xsd:double]];
  ssn:observationResultTime [time:inXSDDatetime "2001-10-26T21:32:52"^^xsd:dateTime];
  ssn:observedBy swissex:Sensor1.

```

**Listing 5.2:** Wind speed sensor observation in RDF

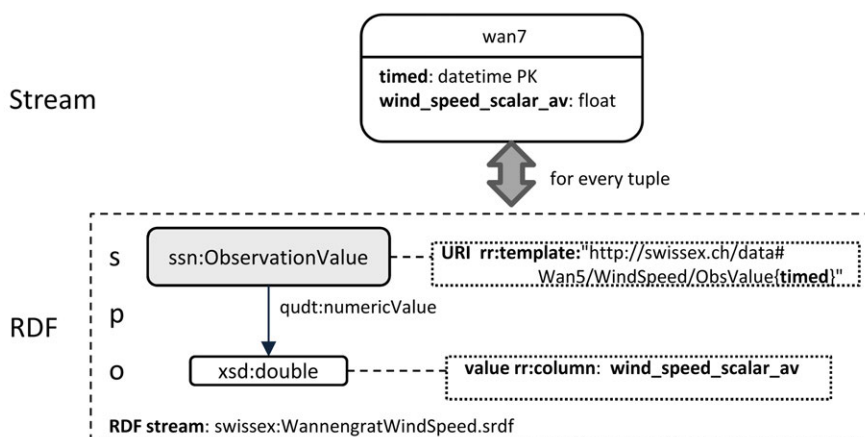
Each observation (e.g. `swissex:WindSpeedObservation1`) is linked to a certain feature of interest, e.g. the wind phenomenon at that location (an instance of `sweetAtmoWind:Wind`, from the SWEET ontologies<sup>1</sup>). Similarly, the `ssn:observedProperty` indicates the property of the feature of interest that has been observed. The observation values captured by the sensor are represented as instances linked to a `ssn:SensorOutput` through the `ssn:hasValue` property. The data values, instances of some data type (e.g. `xsd:double`), can be linked to the observation values using specialized properties from vocabularies that represent quantities (e.g. `qudt:numericValue` from the QUDT ontology<sup>2</sup>).

We can map the observations from the original sensor data schemas, to an RDF representation according to the ontology described above, using the R2RML language. As an example, the following mappings indicate how to generate SSN observation values

<sup>1</sup>In this example the NASA SWEET ontologies are used for representing environmental domain specific properties: <http://sweet.jpl.nasa.gov/ontology>

<sup>2</sup>Quantities, Units, Dimensions and Types: <http://www.qudt.org/qudt/owl/1.0.0/>

from the sensor schemas in Listing 5.5. For every tuple in the wan7 sensor, an instance of the `ObservationValue` class is created according to the R2RML definition in Figure 5.6 (see Listing 5.3, the mappings are expressed themselves in RDF).



**Figure 5.2:** Mapping from the wan7 sensor to a SSN `ObservationValue`.

The mapping definition indicates first from which sensor it will get the data, in this case `wan7`, with the `rr:tableName` property. The triples, each one with a subject, predicate and object, will be generated as follows:

- The subject of all triples will be created according to the `rr:subjectMap` specification. The URI is built using a template (`rr:template` rule), which concatenates a prefix with the value of the `timed` column.
- The subject will be an instance of `ssn:ObservationValue`.
- The triples will belong to the virtual RDF stream `swissex:WannengratWindSpeed.srdf`.
- The predicate of each triple is fixed, in this case `qudt:numericValue`.
- Finally the object will be a `xsd:double`, whose value will be retrieved from the `wind_speed_scalar_av` attribute from the `wan7` stream.

```
:Wan7WindMap a rr:TriplesMap;
  rr:logicalTable [rr:tableName "wan7"];
  rr:subjectMap [rr:template "http://swissex.ch/data#Wan5/WindSpeed/ObsValue{timed}";
    rr:class ssn:ObservationValue;
    rr:graph swissex:WannengratWindSpeed.srdf];
  rr:predicateObjectMap
    [rr:predicateMap [rr:constant qudt:numericValue];
     rr:objectMap [rr:column "wind_speed_scalar_av"]];.
```

---

**Listing 5.3:** Mapping a sensor to a SSN ObservationValue in R2RML

More triple mappings could be specified in a more complex definition, for example including several properties and object instances, not only data values.

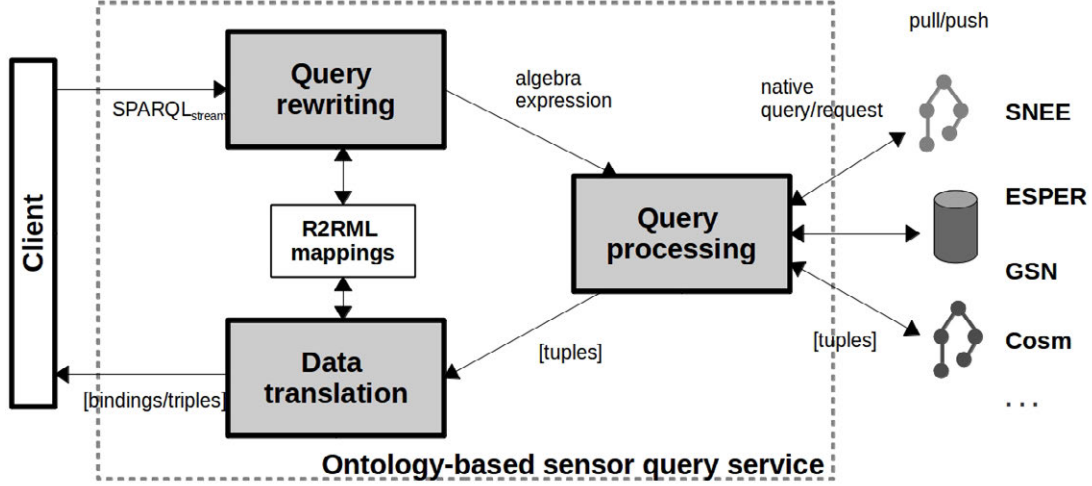
### 5.3 Ontology-based Streaming Data Query Architecture

Having defined an ontological representation for sensor observations, we can query the data values (the actual measurements), using the mappings described in Section 5.2. Since data values are accessible through DSMS, CEP or sensor middleware, we require query rewriting mechanisms to transform our semantic queries (e.g. in SPARQL<sub>Stream</sub>) to queries in terms of the underlying data models. After executing the queries, a data translation process is performed to transform the incoming tuples into triples that will become the final result set.

Our architecture is composed of the ontology-based sensor query service and the data layer (Figure 5.3). The DSMS, CEP or sensor middleware are placed in the data layer, and provide an underlying streaming data querying interface, which is heterogeneous and hidden by the ontology-based query service. This component receives queries specified in terms of an ontology (e.g. SSN ontology network) using SPARQL<sub>Stream</sub> (see Chapter 4). Since the SPARQL<sub>Stream</sub> query is expressed in terms of the ontology, it has to be rewritten into queries in terms of the data sources, using the techniques described in Section 4.5, and the mappings written in R2RML. As a result we expose virtual RDF streams that can be queried with SPARQL<sub>Stream</sub>. The results are either triples or variable bindings that are generated from the original data stream, through a translation process. This process is executed by three main modules: Query rewriting, Query processing and Data translation.

*Query rewriting* uses the R2RML mappings to produce streaming query expressions over the sensor streams. These are represented as the algebra expressions extended with time window constructs, so that logical optimizations (including pushing down projections, selections, and join and union distribution, etc.) can be performed over them (see Section 4.6). These can be easily translated to a target language or stream request, such as a REST API, as we will see later.

*Query processing* is delegated to the DSMS, CEP or middleware, which ingests the translated query or request built from the algebra expression, and can be performed



**Figure 5.3:** Ontology-based sensor query rewriting, processing and data translation.

by explicitly requesting the incoming data from a query (pull) or by subscribing to the new events (triples) that are produced by a continuous query (push). Note that different streaming query processors may have different expressivities and a `SPARQLStream` may not be possible to rewrite to a limited query evaluator.

The final step of *data translation* takes the pulled or pushed tuples from the streaming data engine and translates them into triples (or tuples, depending on whether it is a `CONSTRUCT` or `SELECT` query respectively), which are the final result.

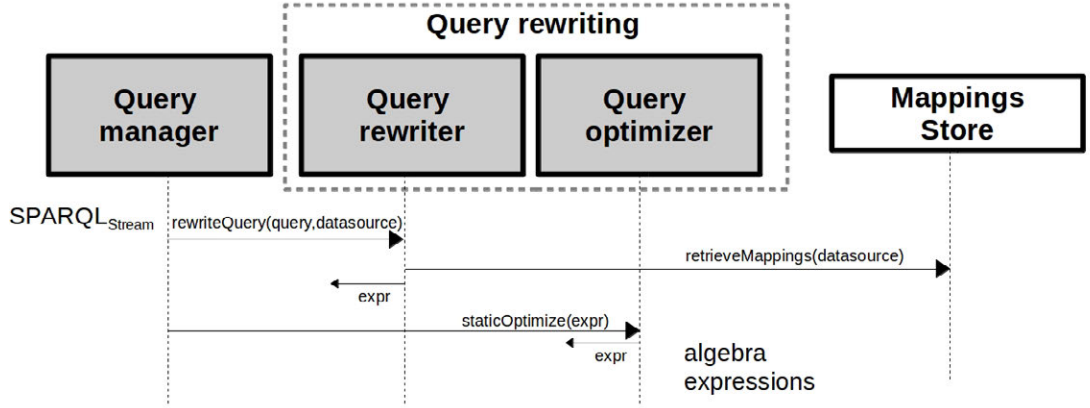
The architecture presented assumes the existence of mappings, and that these are already assigned to the source and the current query to be rewritten and executed. However, in practice there may exist different mappings available, in open or closed repositories that can be maintained by a community. Whichever the case, it is necessary to indicate at some point which is the mapping definition to be used for a given data source.

In our approach, this is achieved by assigning a set of mappings, to a datasource. This datasource identifies a given streaming data source and mapping an R2RML mapping. This operation results in the creation of a virtual RDF stream resource that can be queried with `SPARQLStream`. This constitutes a pre-query configuration step that can be executed once, before multiple `SPARQLStream` queries to the newly created data resource.

Notice that following this approach, it is possible to expose the same data source with different mappings, thus creating a set of virtual resources from the same dataset.

## 5.4 Query Rewriting Process

The query rewriting process of SPARQL<sub>Stream</sub> to algebra expressions, and some of the static optimizations possible at this level have been presented in Chapter 4. While in previous works the task of relating ontology-based queries to streaming query processors has been addressed using implementation specific approaches, we opted for a general specification of abstract semantics that can be later instantiated in concrete languages, depending on the current data source used. Query rewriting is composed of two stages, the rewriting, which uses the mappings associated with the datasource to produce an algebra expression, and the optimization, which further refines this expression using optimization rules. An abstract representation of these steps is depicted in Figure 5.4.



**Figure 5.4:** Rewriting modules: rewriting a SPARQL<sub>Stream</sub> query to algebra expression, and optimizing the expressions.

### 5.4.1 Rewriting to Algebra Expressions

The query rewriting operation takes as an input a SPARQL<sub>Stream</sub> query and a datasource that represents a virtual RDF stream dataset. The result of the operation is an abstract algebra expression, as described in Section 4.5, that uses the mapping used in to create datasource in the configuration step.

At this point the query rewriting process can already detect if the input SPARQL<sub>Stream</sub> query can be answered or not. for example, if the query includes ontology terms that are not present in the mappings, the query cannot be answered altogether. Or, if the data source is not available or has been deleted, the query execution is dropped. The result of this operation is not directly useful to the end-user, as it explains in abstract terms the semantics of the query, as operators over relational streams. This representation is used

for the subsequent steps.

### 5.4.2 Query Optimization

Static query optimization takes an algebra expression `expr` and produces a new one that is semantically equivalent, but after applying the optimization steps described in Section 4.6. As we have already mentioned, this step may simplify the query algebra tree so that it will produce less intermediate results, avoid redundancies in some queries and avoid unnecessary execution of some of the branches of the tree.

Some query processors with limited expressiveness (e.g. sensor middleware) may actually need this type of external optimizations in order to run the queries at all. For instance a processor that does not allow joins may benefit from query optimization when self joins are eliminated or simplified. However, when even after optimization a query is not expressible in the target language, then partial execution of the query can be externalized, and adaptive query optimization can be used in this case.

Although other cost-based and adaptive optimizations can be envisioned after this step, we did not consider them at this stage, as one of the limitations of this work, and appears as an interesting future direction.

## 5.5 Query Instantiation and Delegation

An abstract query algebra expression is instantiated by a Query Adapter specific to the underlying query processor. This operation will produce a `targetquery` in a concrete language.

A query adapter provides the means to plug a new type of query language, that can represent the full or at least a subset of the abstract streaming algebra supported by an RDF streaming query language such as `SPARQLStream`. An execution adapter, in turn, provides the access and protocol specific means to connect to the query evaluator, issue the query and get the results back. At this stage, depending on the available features of the target query, it can be decided whether the `SPARQLStream` query can be answered by the underlying processor. If the target query cannot be instantiated (if it lacks some feature, e.g. joins of streams), then there is no possible execution and the query processing ends.

Different expressivity, query language capabilities and execution models exist in the different target systems, which may result in different query instantiation and delegation mechanisms. We detail the most common cases in the next subsections.



### 5.5.1 DSMS Query processing

In the case of DSMS, the query is instantiated as a declarative query, which in most of the current implementations derives from SQL or the most widely known streaming query languages CQL. The queries are typically evaluated by a data manager (which can be distributed in different sub-managers as well, depending on the implementation), and produce series of results. Depending on the type of query, the results will be available for pull and push delivery, which again, are available depending on the concrete implementation.

DSMS typically execute continuous queries, so the query adapter must allow registering queries that can be periodically executed. To do so, it must keep track of the queries, identified with a `queryid`. Using this identifier, we can retrieve results for the corresponding query, create subscriptions or remove the query from the evaluator. We will detail the query evaluation and data retrieval in Section 5.6.

There is a particular case where the DSMS implements in-network query processing: different nodes of a system are in charge of the execution of a query at the same time that capture the actual data values. This approach emerges from the intuition that querying closer to the data may result efficient, especially in terms of processing, as the network nodes may have limited computational capabilities and scarce energy.

In this case, the `SPARQLStream` query, rewritten and optimized to an algebra expression, will be transformed into a query that can be further decomposed and whose operators may be distributed in the network nodes following cost-based models (Brenninkmeijer et al., 2008). The notion that the data is queried in a distributed fashion and that the actual query operators are performed in the network nodes, is completely hidden from the user.

### 5.5.2 Complex Event Processing

CEP have similar features than DSMS and in fact, some vendors ship products that integrate the functionalities of both of them. Nevertheless, in CEP the focus is on identification of patterns that can be identified as concrete events over time. These events are characterized by means of queries that include patterns. The complexity of these patterns in some CEP may exceed the expressiveness of the corresponding RDF streaming query language, such as `SPARQLStream`. For example, temporal sequences of events are hard to define in `SPARQLStream`, (e.g. indicating that “this event happened just after this other event”). These could also be considered as future extensions for RDF streaming query languages, but for the vast majority of use-cases, the temporal windowing and

streaming of events generated from query expressions are suitable enough.

The possibility of specifying GLAV mappings using R2RML, by declaring logical streams in the left-hand side of the mapping, can lead to using these CEP-specific features in our approach. Because the logical streams in the mappings are completely delegated to the native CEP, our approach can be easily extended in such cases.

### 5.5.3 Middleware Query Processing

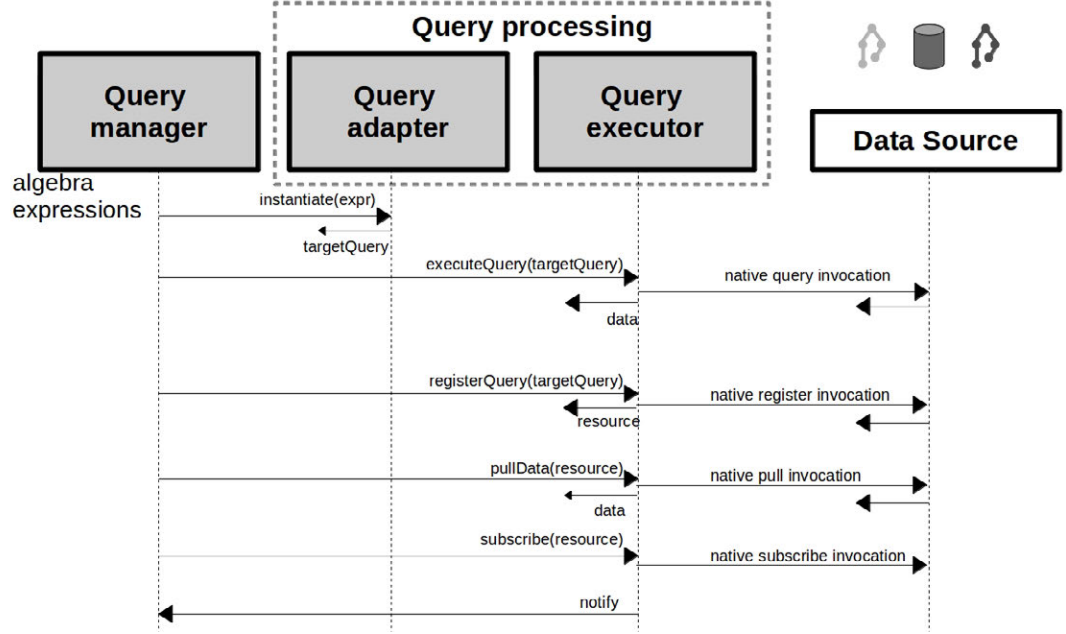
In many real-world scenarios, direct access to databases, DSMS or CEP is limited or restricted so that it is not possible to apply any of the aforementioned processing alternatives. Instead of direct access, however, it is fairly common to provide streaming data in form of services over the web. These are managed by streaming data middleware servers such as GSN or Cosm: the processor used by such middleware is opaque to the service users, they are only provided with an interface with typically very limited query capabilities.

For example, they may only allow receiving the latest values, given the name of a stream. This would roughly correspond to a simple project-all-attributes operation from a stream. Other services may allow projecting specific attributes, creating simple filters or even doing aggregates.

In these cases, query parameters and operators, typically expressed in an ad-hoc syntax, are sent over the network as service parameters. The query adapter for these queries must be aware of these system-specific query forms and be able to instantiate an algebra expression into them. The most common types of these interfaces are SOAP-based services and REST services, which need to be supported by the execution adapter. Also, the results protocol and formats must be supported by the execution adapter, which come usually as XML or JSON instances.

## 5.6 Query Evaluation

After the instantiation and delegation of the query to the underlying DSMS, CEP or middleware, as seen in the previous section, the query is evaluated. This part of the process is almost entirely handed over to the external query processor in our approach, which benefits from the specialized and efficient software available for this purpose. However, from the point of view of the execution adapter, there are different possible interactions, depending on the type of query that is being issued and the type of data retrieval that is required, as seen in Figure 5.5. We discuss these variants in the following subsections.



**Figure 5.5:** Query processing: query instantiation, one-off execution, pull and push data delivery.

### 5.6.1 Snapshot Queries

In the simplest case, the execution adapter launches a one-shot query, also known as “fire and forget”. It is executed strictly once and the results are returned immediately. This is similar to traditional relational database queries, and captures a snapshot view of the data at the moment when it was launched.

In this case the evaluation adapter performs a `executeQuery(targetquery)`, with the `targetquery` parameter representing the rewritten and instantiated query. This results in a snapshot query execution, and its results are immediately returned as tuples. The tuples are transformed, if necessary, to the expected `SPARQLStream` results format, as bindings or RDF, as described in Section 4.7. Notice that the execution adapter is responsible for sending the target query, waiting for results and receiving the results back and managing all the protocol and formatting issues, depending on the type of system that we are dealing with.

### 5.6.2 Continuous Queries

Continuous queries are a powerful way of periodically monitoring the data, looking for patterns and constantly generating results, which is an intrinsic feature of streaming

systems. In this case, the evaluation adapter must first register a continuous query with the `registerQuery(targetquery)` operation. This operation results in the registration of the query in the system, which will be executed according to the query requirements. For instance a window query with a slide of 10 seconds, will recompute a window with that periodicity. The `registerQuery` operation does not return results but only an identifier of a data resource that references the continuous results that will be generated by the query.

In subsequent steps, the evaluation adapter may poll for results using the registered query identifier, or create subscriptions that enable data notifications from the query processor, each time that there are results for the target query.

### 5.6.3 Pull-based Data Retrieval

The results of continuous queries can be pulled periodically, if the client explicitly requests it with a `pullData(resource)` operation. The `resource` parameter is an identifier that represents the continuous query registered with the `registerQuery` operation. Typically, the evaluation adapter will poll periodically with `pullData` requests, which may result in empty results if no data matching the query was produced. Also, the result produced by the continuous query may grow with time, and it is up to the query processor to decide when these results expire or are purged. The client has the ability of Pulling only some subsets of the data, according to the time they were produced, or limiting them by cardinality (e.g. only the latest 10 values, only the latest 10 minutes of values).

### 5.6.4 Push-based Data Retrieval

Instead of pull-based delivery, the client may request a subscription to the results of the continuous query, by emitting a `subscribe(resource)` operation. In this case the query processor will send results to the subscriber each time there is data matching the query. This is done via a `notify` operation sent every time there are relevant results. This is a convenient way for the evaluation adapter to handle real time result delivery, although it can be complex to implement in web scenarios where publisher-to-subscriber communication is complicated.

## 5.7 Implementing Ontology-based Streaming Query Rewriting

The architecture presented in the previous section, has been implemented for different DSMS, CEP and sensor middleware, as we describe in the following sections.

### 5.7.1 Query Processing

In the previous section we described the query rewriting process, which generates algebra expressions that are generic and can be serialized in different query languages. While in previous works the processing was limited to specific platforms (Calbimonte et al., 2010b), we have now implemented adapters for systems with different characteristics, namely SNEE (DSMS), Esper (CEP), GSN and Cosm (middleware), showing the generality of our approach.

---

```

SELECT ?windspeed ?tidespeed
FROM NAMED STREAM <http://swiss-experiment.ch/data#WannengratSensors.srdf>
[NOW-10 MINUTES TO NOW-0 MINUTES]
WHERE {
    ?WaveObs a ssn:Observation;
             ssn:observationResult ?windspeed;
             ssn:observedProperty sweetSpeed:WindSpeed.
    ?TideObs a ssn:Observation;
             ssn:observationResult ?tidespeed;
             ssn:observedProperty sweetSpeed:TideSpeed.
    FILTER (?tidespeed<?windspeed)
}

```

---

**Listing 5.4:** SPARQL<sub>Stream</sub> query requesting wind speed higher than tide speed.

As an example, we will consider the SPARQL<sub>Stream</sub> query in Listing 5.4, which incorporates time windows, filters and two different observations. Suppose that we have mappings that relate the SSN-based ontology network concepts to the streams wan7 and wan6 of Listing 5.5.

---

```

wan7: {wind_speed_scalar_av FLOAT,timed DATETIME}
wan6: {tide_speed FLOAT,timed DATETIME}

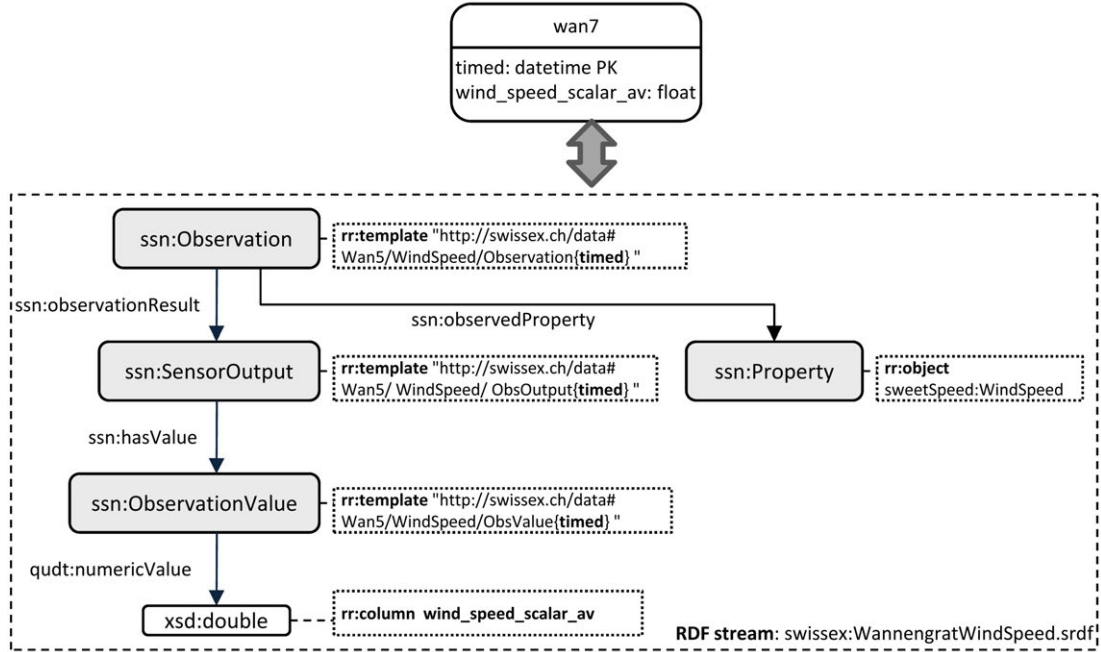
```

---

**Listing 5.5:** Wind and tide speed sensor stream schemas

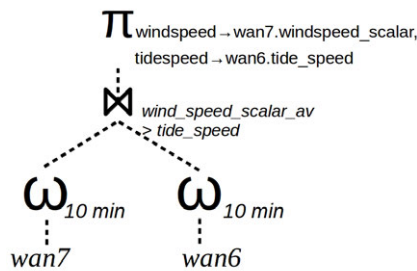
We show a sample mapping for the wan7 sensor in Figure 5.6. This mapping generates not only the ObservationValue instance but also a SensorOutput and an Observation for each record of the sensor wan7. Notice that each of these instances constructs its URI with a different template rule and the Observation has an observedProperty property to the sweetSpeed:WindSpeed property. Suppose a similar mapping for wan6, only that has sweetSpeed:TideSpeed as observed property.

The query translation process will use these mappings to generate an algebra representation. As the query terms are matched against the mapping definitions, both sensors



**Figure 5.6:** Mapping from the wan7 sensor to a SSN Observation.

are included in the expression, first applying the window to both sensors, and projecting the required fields to build the URIs and values. Then both are merged in a join, as they are both part of the query, but the join includes the condition that compares the values of the wave and tide speed. The result is depicted in Figure 5.7. This can be later serialized into a query and executed by a query engine.



**Figure 5.7:** Algebra expresison after query rewriting.

The query engines may accept query languages or requests through APIs, and in both cases are straightforward to represent as the expressions discussed above. In the following subsections we discuss four implementations of this approach: the SNEE DSMS, the sensor web middleware platforms GSN and Cosm, and the complex-event processor

Esper.

**SNEE.** SNEE (Galpin et al., 2009) is a streaming data query engine able to integrate stored relational sources and capable of in-network query evaluation, delegating parts of the query plan to the sensor nodes. It uses the SNEEqL language, which has a well defined semantics for queries over event streams, acquisitional streams and stored data. Constructing SNEEqL sentences from the algebra expressions we showed in the previous sections is straightforward. For example, the query in Listing 5.6 is produced for the expression in Figure 5.7:

---

```
SELECT wan7.wind_speed_scalar_av AS windspeed, wan7.timed AS windts,
      wan6.tide_speed AS tidespeed, wan6.timed AS tidets
FROM   wan7[FROM NOW-10 MINUTES TO NOW], wan6[FROM NOW-10 MINUTES TO NOW]
WHERE  wan7.wind_speed_scalar_av>wan6.tide_speed
```

---

**Listing 5.6:** SNEEqL translated query

Although SNEE is interesting from the point of view of its ability to perform in-network query processing, some features like union of windows or joins between streams are only partially supported in the current prototype.

**Global Sensor Networks (GSN).** In the GSN sensor middleware, GSN server instances can be queried through web-services or RESTful URL interfaces. The ontology-based sensor query processor can generate GSN API<sup>1</sup> URLs from the algebra expressions, which are executed by the GSN server. Back to our example in Figure 8, the GSN URL API does not support joins between streams, unless there is a virtual sensor that already joins them (complex queries can be defined with virtual sensors in GSN through configuration). Therefore the query is not translatable, but it can be split into two simpler queries and then join the results. We show one of this simpler SPARQL<sub>Stream</sub> queries (Listing 5.7) and its translation to a GSN URL. We used this implementation in the Swiss-Experiment use case of Section 7.2.

---

```
SELECT ?windspeed
FROM NAMED STREAM <http://swiss-experiment.ch/data#WannengratSensors.srdf>
[NOW-10 MINUTE TO NOW-0 MINUTE]
WHERE {
  ?WaveObs a ssn:Observation;
  ssn:observationResult ?windspeed;
```

---



---

<sup>1</sup><http://sourceforge.net/apps/trac/gsn/wiki/web-interfacev1-server>

### 5.7. Implementing Ontology-based Streaming Query Rewriting

---

```
ssn:observedProperty sweetSpeed:WindSpeed.  
}
```

---

#### **Listing 5.7:** Simplified SPARQL<sub>Stream</sub> query

#### **Listing 5.8:** Generation of a GSN API URL

---

```
http://montblanc.slf.ch:22001/multidata?vs[0]=wan7&  
field[0]=wind_speed_scalar_av&  
from=15/05/2011+05:00:00&to=15/05/2011+15:00:00
```

---

**Cosm.** While GSN is used in several projects and research initiatives, other wide-open sensor data systems are emerging, such as Cosm, which offers data management of real-time data from sensors. The data hosted in Cosm is organized as tagged environments or feeds, each one having one or more datastreams, which represent an individual measuring device, and the actual values, called datapoints. Cosm data can be queried through a RESTful API, although the complexity of these queries is low, compared to those in GSN. For instance, it is not possible to perform joins nor selections or aggregates, but it remains an interesting data source for open and large-scale use. The API allows retrieving the latest datapoints of a certain datastream, or the datapoints in a time interval specified as part of the request. For example the following request: <http://api.pachube.com/v2/feeds/14321>, returns data from the environment with id=14321, including the list of its data streams. A time-based query can be specified for a particular datastream, like in the following example:

---

```
http://api.pachube.com/v2/feeds/14321/datastreams/4?start=2011-09-02T14:01:46Z&end=2011-  
09-02T17:01:46Z
```

---

#### **Listing 5.9:** Cosm request

In this case the datastream has an id=4, and the time boundaries are given by the start and end parameters. To query these Cosm streams with our approach, we specify the environment id as the stream name in the R2RML mapping, and the datastream id as an attribute name.

**Esper.** Esper is a commercial event processing engine that supports streaming data and continuous queries. It provides a rich declarative query language, EPL, with support for a number of streaming data operators, including time windows. Although the query syntax is slightly different from SNEEqL or CQL, the ideas are similar. For instance, Listing 5.10 is the translated EPL query for the Listing 5.7 expression.



---

```
SELECT wind_speed_scalar_av, timed FROM wan7.win:time(10 min)
```

---

**Listing 5.10:** EPL translated query

One of the features of Esper is that it supports both pull and push based delivery of query results. While all the previous implementations we explored dealt only with pull mechanisms, we implemented a push adapter for Esper. In fact the query rewriting phase as such does not change at all (only the serialization to the EPL syntax has to be handled, as mentioned above), but it is mainly the data translation phase that changes. Each time that Esper notifies about a new event, this data is translated to tuples (or triples) and a new event is raised to the subscriber, containing the new tuples (or triples) as argument.

## 5.8 Discussion

We presented an architecture for ontology-based querying for streaming data sources using SPARQL<sub>Stream</sub>. Our approach is based on the query rewriting techniques introduced in Chapter 4, which requires mappings from streams to ontological concepts. We proposed using R2RML for representing such mappings, and showed feasibility evidence for a wide range of existing streaming data engines such as DSMS, CEP and sensor middleware.

Our approach can be compared to existing systems that enable streaming or time-aware SPARQL query provision. We summarize the characteristics of SPARQL<sub>Stream</sub> compared to these systems in Table 5.1. Of these  $\tau$ SPARQL and TA-SPARQL are not designed for continuous processing, and as such do not meet the needs of streaming data processing. CQELS implements an RDF stream native query processor, but as such it re-implements functionality already existent in already available processors, and cannot be plugged to existent streaming engines, unless through an adapter which would incur in overhead. EP-SPARQL is also an implementation from scratch, although based in logic programming. C-SPARQL also relies in an internal DSMS, although it is limited to certain implementations and does not provide flexible mapping-based rewriting, but fixed schemas.

Although SPARQL<sub>Stream</sub> provides flexibility in the number of supported underlying platforms, based on declarative mappings and query rewriting, it comes at a cost. This

## 5.8. Discussion

is mainly due to the overhead of query rewriting and data translation from the incoming data source. The cost is highly variable depending on the type of data model, format and protocol used to communicate with the DSMS, CEP or middleware associated. Also, the expressiveness of the streaming data language may limit the range of SPARQL<sub>Stream</sub> queries that can be rewritten and executed in the target processor.

Language	Input	Execution	Query Optimization	Continuous execution	Stored data	Reasoning
TA-SPARQL	Transform to TA-RDF	translation to SPARQL, time indexing of stored data	No	No	-	No
τSPARQL	Conversion to τRDF	SPARQL evaluation, using keyTree time indexed graphs	No	No	-	No
Streaming SPARQL	RDF stream	physical stream algebra	Static plan optimization	Yes	Yes	No
C-SPARQL	RDF stream	DSMS based evaluation of streams + triple store for static RDF	Static plan optimization	Yes	Yes (Internal Triple store)	RDF entailment, incremental materialization
CQELS	RDF stream	RDF Stream processor	Adaptive query processing operators	Yes	Yes (Stored Linked Data)	No
SPARQL <sub>Stream</sub>	relational stream	external query processor	Static algebra optimizations, host evaluator specific	Yes	Yes (datasource Dependent)	No
EP-SPARQL	RDF stream	logic programming, backward chaining rules		Yes	Yes	RDFS, Prolog equivalent

**Table 5.1:** Comparative table of SPARQL<sub>Stream</sub> ontology-based system and alternative approaches.

Finally, we note that some of the approaches developed concurrently to SPARQL<sub>Stream</sub> have additional features, that could also be incorporated in the future. For instance sequencing patterns, or reasoning. While sequencing may require only a corresponding operator in the target language, reasoning needs meta-level knowledge, as explored in query rewriting-based reasoning for SPARQL (Pérez-Urbina et al., 2009). An logical trend for these SPARQL extensions would be to converge towards a common standard, although for different needs, the very distinct types of implementations may remain necessary.



## Chapter 6

# Sensor Metadata and Data Characterization

With more and more sensors deployed every day, and the increasing number of its applications, it becomes expensive and difficult to manage the data produced by these numerous and heterogeneous sensor nodes. An example of this scenario is the Swiss Experiment<sup>1</sup>, a multi-disciplinary project and platform that enables real-time experiments through a large-scale federation of environmental sensor networks. The data collected from the Swiss Experiment sensors includes measurements of *temperature*, *CO<sub>2</sub>*, *moisture*, etc., mainly in the Swiss Alps. However, the data is heterogeneous as it comes from different geographical locations, with different time spans (e.g. observations collected during 1 year, 3 months, etc.), as well as varying sampling rates (e.g. per minute, per 10 minutes). Moreover, the metadata for these sensor types is not always complete and coherent. As an example, to indicate that a sensor measures temperature (i.e. the *observed property*), different sensors use various tag names, like “temperature”, “temp”, “t”, “msptemperature”, “tp”, etc. These user-generated pieces of metadata are inherently noisy, and constitute a big challenge for sensor data management and analysis, because the semantics of the data are hidden behind these textual tags. This is a major obstacle for any correlation, search, aggregation or fusion task on sensor web data.

In less-controlled scenarios than the Swiss Experiment, the problems of heterogeneity are even more noticeable. For instance in the Cosm web platform, users also use tags as metadata for their sensor streams, identifying which types of measurements they are

---

<sup>1</sup>Swiss Experiment: <http://www.swiss-experiment.ch/>

publishing. Projects like the Air Quality Egg<sup>1</sup>, aiming at promoting air-quality participatory sensing, enable almost any citizen to publish measurements at web-scale. However, the user-provided metadata is often incomplete. For instance, a tag may denote a location (e.g. “madrid”), or an observed property (e.g. “co2”) or even a project name (e.g. “airqualityegg”) or a unit of measurement, etc. In many cases these tags are misleading or they are not provided at all, making it very hard for other users to query or make use of this data.

Even though there are techniques for representing semantically-enriched sensor metadata using standardized vocabularies and ontologies (Compton et al., 2009a), these often require manually mapping or annotating the sensor streams. However, many of the sensor metadata properties can be extracted from the actual data values of the sensor data sources, by looking at patterns and behavior of the sensor time series. In this chapter we propose an approach for sensor data analysis that infers semantic properties such as the type of observed property, using the raw sensor observations as input. We summarize the main contributions of our approach as follows:

- We propose novel method for representing time series as distributions that represent the slopes of a linear approximation of the initial numeric sensor measurements.
- Based on the statistics of the observation slopes, we infer the type of observed property of the sensor measurements. We use a supervised classification method that exploits the similarity of the slopes distributions. Additionally, we study the classification using smaller subsets of the data.
- We provide a mechanism for enriching sensor metadata, based on the SSN Ontology (Compton et al., 2012), with the metadata inferred from the observation slopes.
- We build an architecture for linking raw sensor measurements to high-level semantics, and validate our method using two real-life environmental sensor datasets.

## 6.1 From Raw Measurements to Semantic Metadata

Sensor data is typically represented as time series, describing the evolution over time of a certain observed property. Raw sensor data without any metadata that describes

---

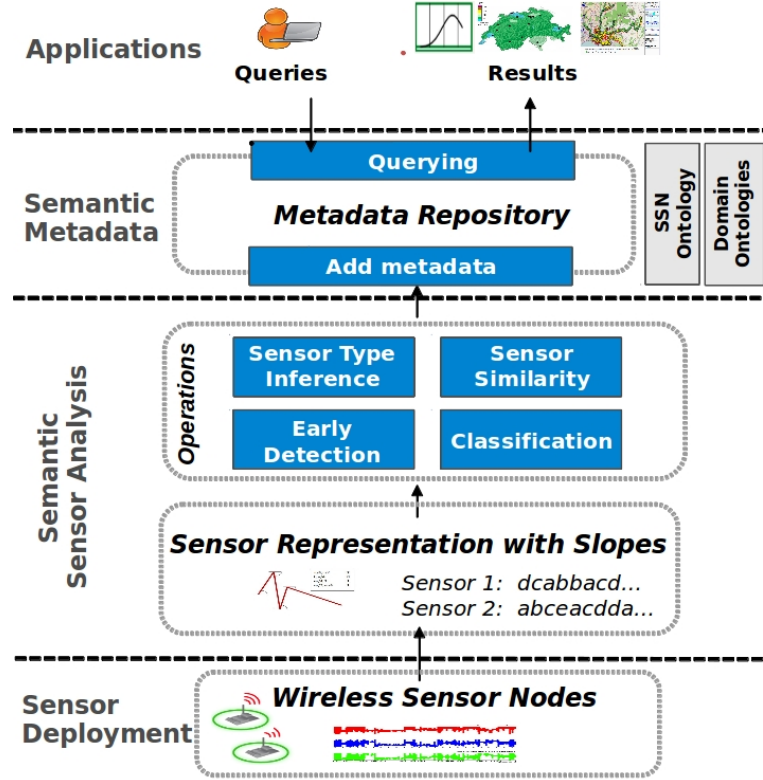
<sup>1</sup>AirQuality Egg <http://airqualityegg.wikispaces.com/>

it, has limited use as it is hard to discover, integrate or interpret. While in controlled environments the sensor metadata can be reasonably well managed and controlled by the data owners, in the context of the sensor web, where any citizen is able to produce and publish data, it becomes a more difficult task. While semantic metadata has been shown to be effective for managing large sensor metadata repositories (Babitski et al., 2009; Henson et al., 2009; Wei and Barnaghi, 2009), current proposals require expensive manual curation and tagging (Calbimonte et al., 2011d). However, these approaches do not look into the data values, from which we can derive some of these metadata properties using analysis and mining techniques.

We depict in Figure 6.1 our architecture for deriving semantic metadata from sensor data measurements. The approach includes characterizing sensor time series and extracting their observed property types to enrich sensor metadata, and consists of four main layers:

- At the *sensor deployment* layer, sensor nodes provide initial measurements in terms of real-time numerical values, e.g. air temperature, relative humidity, etc., in terms of a given unit of measurement. These measurements are collected as streams of data values.
- In the *semantic sensor analysis* layer, we first use linear approximations to represent subsets of the sensor data streams, and calculate the slopes of the linear segments, from which we derive slope distributions. Based on the sensor slopes, we are able to compute similarity between sensor data series, inferring the observed property types through classification. Our approach needs only a small partial subset of the data to characterize a data stream.
- From the analysis layer, we integrate the new inferred information into the *semantic metadata*. Using the SSN Ontology as a basis, and combined with domain specific ontologies, this enriched metadata is made available for further processing, querying or reasoning.
- In the application layer, users can build tools and visualizations to query such sensor data and receive results that include the new metadata computed by the analysis layer.

The deployment layer is usually built using sensor or stream data management systems. These systems centralize the data captured by the devices and provide storage,

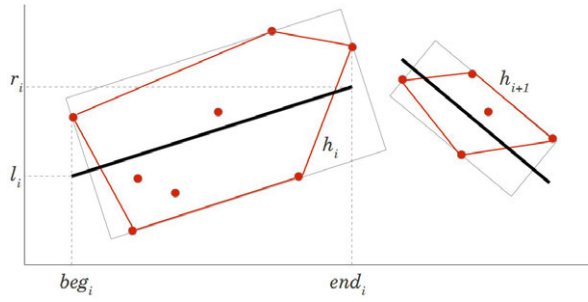


**Figure 6.1:** An architecture for inferring semantic sensor metadata from raw measurements.

query interfaces and streaming operators. An example of these semantic streaming query interfaces is  $\text{SPARQL}_{\text{Stream}}$ , described previously in Chapter 4. As for the semantic metadata, we built upon previous work on semantic management of sensor networks (Calbimonte et al., 2011d), centered on the use of the SSN Ontology (see Section 2.4.2 for details), coupled with domain ontologies and vocabularies for quantities and units of measurements. For the analysis of the time series, we propose a representation based on the slopes of a linear approximation of the data, as described in Section 6.2. Then these representations can be used to compare and find similarities among new and existing time series, classifying them according to the detected observed property type, etc. As a result, we are able to complete and query the sensor metadata, as detailed in Section 6.3.

## 6.2 Representing Sensor Data with Segment Slopes

In some types of sensor data, such as environmental time series, similar patterns can be observed periodically over time. These patterns can be characteristic to a type of sensor data, and therefore may help an observer to recognize it. If we represent a time series using a linear representation, such as the one in Figure 2.9, the patterns of the data can be associated to the angles of the linear segments or its corresponding slope. For instance, a steep slope indicates a sudden increase of the measured property. The intuition is that if these slopes are repetitive over time, we can build slope distributions that can be representative of a type of time series. Using slopes makes it possible to find similarities between time series that not necessarily have the same value ranges but similar behavior, e.g such as the *air temperature* in two different locations.



**Figure 6.2:** Piecewise linear approximation, construction of the convex hull for the points comprised between  $beg_i$  and  $end_i$ .

### 6.2.1 Background: Piecewise Linear Representation

As seen in Section 2.5, we can use linear segments to approximate a time series (Piecewise Linear Representation, PLR), and analyze the trends by observing the angles that the segments form (Section 2.5.1). Notice that the number of points for a segment can be variable (adaptive approximations). We used the algorithm of (Buragohain et al., 2007) for the construction of piecewise linear histograms.

Consider we have a time series of  $n$  data points  $X = x_1, x_2, \dots, x_n$ , and we want to fit it in  $m \ll n$  segments. The algorithm maintains a set  $B$  of buckets  $b_i = h_i, beg_i, end_i, l_i, r_i, h_i$ , where  $h_i$  is a convex hull of data points, and  $(beg_i, l_i), (end_i, r_i)$  are the coordinates of the segment that best fits the convex hull (the segment that bisects the thinnest bounding rectangle of  $h_i$  (Buragohain et al., 2007)). The slope of  $b_i$  can be calculated as  $slope(b_i) = \frac{r_i - l_i}{end_i - beg_i}$ . The algorithm adds elements to  $B$  from  $X$ , until there are no buckets available, and then it starts to merge those adjacent buckets  $b_i$  and  $b_{i+1}$  that



combined produce the smallest increase in total error. Merging is reduced to a convex hull merge of  $h_i$  and  $h_{i+1}$ . The algorithm iterates until all elements of  $X$  have been placed in a bucket. The resulting set of segments of each bucket  $b_i$  is the linear approximation of  $X$ .

For instance in Figure 6.2, the convex hull  $h_i$  encloses 8 data points and its minimum rectangle is bisected by the thick black segment defined by the points  $(beg_i, l_i), (end_i, r_i)$ . This is the linear representation for these 8 points. During the computation of the linear representation, if merging  $h_i$  with the next hull  $h_{i+1}$  reduces the approximation error, they will form a new single hull with its own bisecting segment. Once we apply this PLR algorithm we have the time series represented as line segments, each with a distinctive slope.

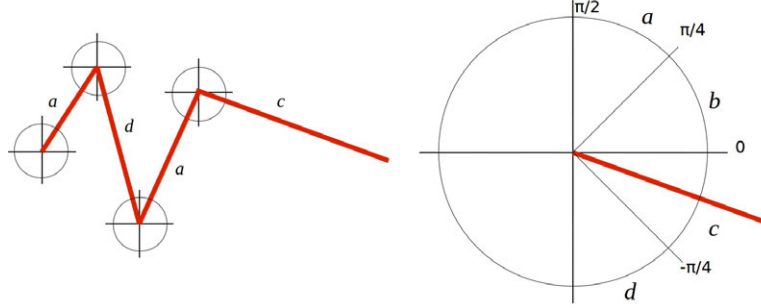
### 6.2.2 Slope Distributions

To build the slope distributions, we first compute a linear approximation of the time series, using the algorithm described in Section 6.2.1. It is possible to create linear approximations of different accuracy, depending on the number of segments per unit of time. For instance for a time series of 30 days, if we use 4 segments per day, their slopes will reflect coarse-grained changes in the data during each day. Time series of originally different sampling times, can be represented using the same segment/day rate, in order to be comparable. Obviously, if the original sampling interval is greater than the number of segments/day, the representation with that rate is not possible.

Once the linear representation is built, we can compute the slopes and analyze them. The slope or gradient space, bounded in the  $[\infty, -\infty]$  interval for the possible angles  $[\frac{\pi}{2}, -\frac{\pi}{2}]$ , can be divided in sectors, each represented with a symbol  $\alpha_j$  from an alphabet  $A$  and we can assign each segment to its corresponding symbol. We propose using the segment representation discussed in the previous section, to compute *slope symbolizations*, which characterize a time series as a sequence  $S$  of symbols  $s_i$  from an alphabet  $A$  that correspond to a type of slope.

In this way, we characterize a time series by the type of variations present in the sensor data, regardless of the data values. For example if we divide the angle space in 4 sectors (labeled  $a, b, c, d$ ), at intervals of  $\frac{\pi}{4}$ , we can match each segment slope with one symbol. For instance in Figure 6.3 we have 4 segments, whose symbolic representation is  $adac$ , by matching each slope with a symbol.

Having this symbolic representation of the slopes, it is possible to compare them



**Figure 6.3:** Slopes symbolization. The angle space in this example is divided in 4 sectors, each of  $\frac{\pi}{4}$ . According to which division the segments falls in, it is assigned a symbol.

to check if two series have similar slope patterns. One simple way to do so, is to generate *symbol distributions*, or histograms that count how many symbols of each type exist in a time series. So a distribution of a sequence  $S$  can be defined as a set  $D_S$  of elements  $d_{\alpha_j} = |\{s_i \in S, s_i = \alpha_j\}|$ , for all symbols in  $A$ . For the previous example, it would be a vector 2,0,1,1, which can be normalized by the total elapsed time, so that we can compare series encompassing different time spans. A simple distance measure is the euclidean distance, defined for two distributions  $D_{S_1}, D_{S_2}$  of length  $n$  as:

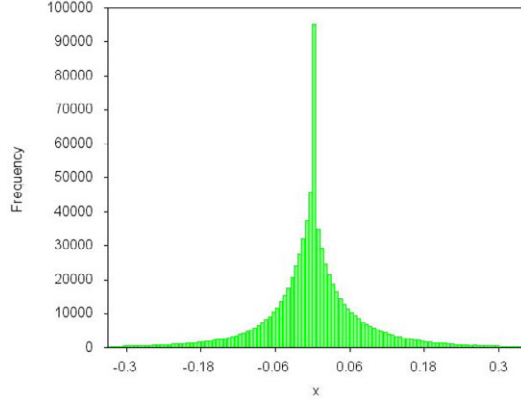
$$d_{eucl}(D_{S_1}, D_{S_2}) = \sqrt{\sum_i^n (d_{S_1i} - d_{S_2i})^2}$$

While most symbolizations are computed using the magnitudes of the linear segments, we use the angles, because these provide information about the trend of the time series, regardless of the scale.

### 6.2.3 Choosing the angle divisions

Although we can arbitrarily choose how to divide the angle space (e.g. 4 sectors of  $\frac{\pi}{4}$  as in the previous example), the actual angles may be more concentrated in some intervals than others. For instance time series with highly changing angles such as *wind speed*, may have steeper gradients than a more stable series. Taking into account this fact, we propose to analyze the training data sets to determine an angle division that better represents the actual distribution of angles in the training set. We opted for dividing the angle space such that every division holds the same number of angles of the training data. For instance, we plotted in Figure 6.4 a histogram that represents the distribution of angles in a training dataset of AEMET meteorology time series. As it can be seen the angles are concentrated near 0 and are scarcer when approaching to  $\frac{\pi}{2}$  and  $-\frac{\pi}{2}$ .

Using this distribution information, we can divide the angle space in divisions that



**Figure 6.4:** Distribution of the angles between  $[-\frac{\pi}{2}, \frac{\pi}{2}]$ , for the AEMET training set.

hold the same number of angles of the training data. Notice that we can choose the number of divisions as an input parameter.

## 6.3 Deriving Semantic Metadata

After establishing how the data is segmented and symbolized, we can use the symbol distributions for data analysis tasks to help understanding the semantics of the data. Given a time series, if it does not contain appropriate metadata, the potential user of this data can use already analyzed time series with their metadata and compare the new one with them. We show how this can be done using our symbolization and a simple classification scheme, even with a partial subset of a time series.

### 6.3.1 Semantic Descriptions

A semantic description of an observation is a collection of statements that includes the observed property (e.g. humidity, pressure), feature of interest (e.g. the air at some location), unit of measurement, among others. For instance, using the vocabulary of the SSN Ontology [Compton et al. \(2012\)](#), we describe a *wind speed* observation in Listing 6.1. The observation, identified as `swissex:WindSpeedObservation1`, has been observed by sensor `swissex:SensorWind1` and reported a value of 6.245. The sensor observed property type `cf-property: wind_speed` (speed of the wind feature) is defined in a domain specific vocabulary (in this case the Climate and Forecast vocabulary defined by the W3C SSN-XG group<sup>1</sup>). Additional metadata about this observation are omitted for brevity.

<sup>1</sup>C&F vocabulary: <http://purl.oclc.org/NET/ssnx/cf/cf-property>

### 6.3. Deriving Semantic Metadata

---

```
swissex:WindSpeedObservation1 rdf:type ssn:Observation;  
  ssn:featureOfInterest cf-feature:wind;  
  ssn:observedProperty cf-property:wind_speed;  
  ssn:observationResult  
    [rdf:type ssn:SensorOutput;  
      ssn:hasValue [qudt:numericValue "6.245"^^xsd:double]];  
  ssn:observedBy swissex:SensorWind1;
```

**Listing 6.1:** Wind Speed observation in RDF according to the SSN ontology

Concretely, the `cf-property:wind_speed` property indicates that this is an observation of wind speed, and it has further semantic information in the Climate & Forecast ontology, as seen in Listing 6.2. It states that it is a property of the wind (`cf-feature:wind`) and is a property of the more general *speed* quantity (`qu:speed`). In order to extract this information, the type of observed property from an unannotated dataset, we propose the classification scheme in the next subsection. The goal is basically to identify the `ssn:observedProperty` for a time series.

```
cf-property:wind_speed rdf:type dim:VelocityOrSpeed;  
  rdfs:label "wind speed";  
  ssn:isPropertyOf cf-feature:wind;  
  qu:propertyType qu:scalar;  
  qu:generalQuantityKind qu:speed.
```

**Listing 6.2:** Wind Speed property according to the Climate and Forecast vocabulary

#### 6.3.2 Data Classification

Given two sets of time series, a *training set* already annotated according to the type of data that is captured, and an unannotated *test set*, we are interested in finding the observed property for the second set. Assume we have a collection  $\mathcal{D}$  of symbol distributions  $D_1, \dots, D_i, \dots, D_n$  as a training set, each of them corresponding to a time series  $ts_i$ , already classified with a type observed property (e.g. “wind speed”). The classification task consists in finding the best property for time series  $ts_{test}$  in the test set.

We can use a simple k-nearest neighbor scheme, which has been successfully used for time series classification [Lin et al. \(2007\)](#); [Xing et al. \(2010\)](#). First, the time series  $ts_{test}$  is segmented and symbolized. Then, we generate a symbol distribution  $D_{test}$ , as described in Section 6.2.2, which can be compared iteratively with each of the distributions  $D_i$  in  $\mathcal{D}$ . From the  $k$  distributions closer to  $D_{test}$ , we select the observed property of the majority.

### 6.3.3 Using Partial Data Subsets

This classification technique may use all the complete time series for computing the symbolization and the slope distribution. However, for types of data with recurring patterns such as the ones present in environmental and meteorological data, using a smaller subset of data can be enough to extract the feature that helps detecting the type of observed property. In that case for the construction of the linear representation of the data, we simply choose a subset of the original data:  $X = x_1, x_2, \dots, x_n$ , with a different  $n'$  such that  $n' < n$ .

### 6.3.4 Querying using the Analysis Results

After executing the classification, we can use the extracted information to complete the sensor metadata, that is then available for querying. In Listing 6.4 we show a simple SPARQL query that asks for sensors that measure air temperature.

```
SELECT ?sensor
WHERE {
  ?sensor a ssn:Sensor;
         ssn:observes cf-property:air_temperature.}
```

**Listing 6.3:** Query all sensors that measure air temperature

The streams produced by sensors can be seen as streaming datasets, whose metadata can also be queried. The stream, identified by a URI, can be seen as an unbounded dataset of observations, some of which are actually used to compute the slope symbolizations and classification described above. The observed properties obtained for the sensor (e.g. cf-property:air\_temperature) are therefore the observed properties of the stream observations. We can also query for more general types of data, for instance, the generic *temperature* property. In Listing 6.4 we ask for all stream URIs of sensors that measure some type of temperature.

```
SELECT ?stream ?observedProperty
WHERE {
  ?sensor a ssn:Sensor;
         ssn:observes ?observedProperty.
  ?stream ssn:isProducedBy ?sensor.
  ?observedProperty qu:generalQuantityKind qu:temperature.}
```

**Listing 6.4:** Query all streams of sensors that measure air temperature

Furthermore, we can expose the similarity measurements computed between the time series, so that users can also query this information. As an example, in Listing 6.5

we use the Similarity Ontology<sup>1</sup>(sim) to represent the computed distance between two series, using our slope representation. Then we can query, for instance the top 5 series similar to a given time series.

```
swissex:slopeSim1.2 a sim:Similarity;
  sim:subject swissex:timeseries1;
  sim:object  swissex:timeseries2;
  sim:weight  0.32;
  sim:method  swissex:SlopeDistributionDistance.
```

**Listing 6.5:** Slope distribution similarity between two time series

This type of queries allows users not only to use the final results of a classification task, but also to query more detailed information including the precision of the computations. This information can be used to validate this metadata or provide insight about the analysis process and the relationship of a sensor stream with other streams. In the case of the early detection of the observed property of a time series, the user may be interested in knowing, for example, how many days of data are typically used for classifying those sensors that measure wind speed 6.6.

```
SELECT ?sensor ?dur
WHERE {
  ?sensor a ssn:Sensor;
          ssn:observes cf-property:wind_speed.
  ?timeseries ssn:isProducedBy ?sensor.
  ?timeseries swissex:duration [qu:numericalValue ?dur].}
```

**Listing 6.6:** Query the number of data days used for classifying wind speed sensors

## 6.4 Discussion

We presented in this chapter an approach for characterizing potentially unknown sensor data streams. In particular we identify the type of data from sensor data sources, using a symbolic representation of the time series slopes. We have shown how this representation can be used for enriching semantic sensor metadata. We have shown specific use cases of time series data classification, providing similarity measures, and metadata aggregation that can be queried in terms of high-level standard ontologies. In Section 8.4 we will show our evaluation of this approach with real-life datasets of the Swiss-Experiment project and AEMET, in terms of precision and recall.

---

<sup>1</sup>The Similarity Ontology: <http://purl.org/ontology/similarity/>



## Chapter 7

# Experimentation with SPARQL<sub>Stream</sub>

The query rewriting and data translation approach described in Chapter 5, using the SPARQL<sub>Stream</sub> query language detailed in Chapter 4, has been successfully used in several projects and implementations. Each of these projects presented different requirements and technological challenges. We discuss three of them in the following sections: SemSorGrid4Env, Swiss Experiment and Ciudad2020.

### 7.1 SemSorGrid4Env

The main objective of the SemSorGrid4Env<sup>1</sup> project was to specify, design, implement, evaluate and deploy a service-oriented architecture and middleware which allows application developers to build semantic-based sensor network applications for environmental management (Gray et al., 2011a).

One of the specific goals of SemSorGrid4Env was to design and implement an ontology-based data integration service for streaming and stored data. The integration service interacts with other components of the SemSorGrid4Env architecture (Gray et al., 2011a) through web service interfaces (see Figure 7.1).

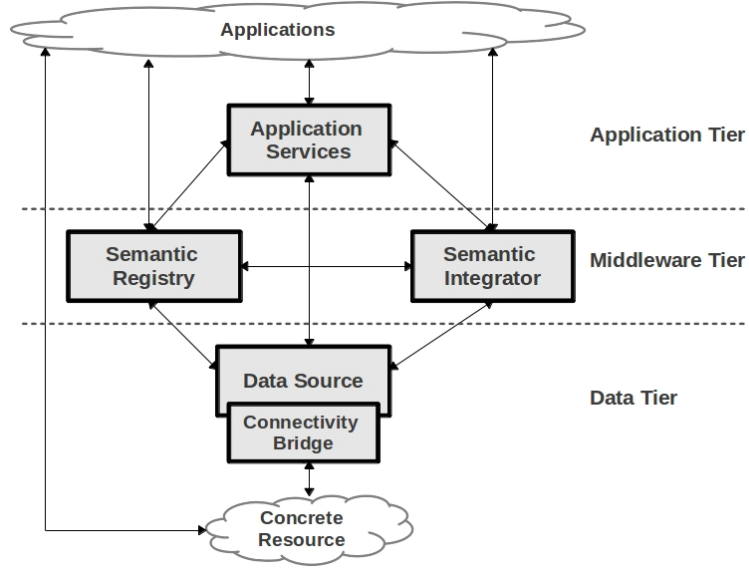
The resulting Integration and Query Service (IQS) provides the following functionality:

- Create an integrated data resource exposed through an ontological schema, given a set of streaming data resources and a mapping document.

---

<sup>1</sup>Semantic Sensor Grid for Rapid Application Development for Environmental Management, FP7: <http://www.sensorsgrid4env.eu>.





**Figure 7.1:** SemSorGrid4Env architecture components (Gray et al., 2011a).

- Query integrated data resources using a SPARQL extension for streaming data (SPARQL<sub>Stream</sub>), while the service internally accesses the underlying data sources.
- Provide data access to the continuous data queries registered for the integrated data sources.
- Register the created data resources in the SemSorGrid4Env Registry (Karpathiotakis et al., 2011; Koubarakis and Kyzirakos, 2010).
- Provide metadata for the integrated data resources in the form of property documents as specified in the architecture (Gray et al., 2010).

The system developed consists of two main components. The first is a software library<sup>1</sup> that is capable of translating SPARQL<sub>Stream</sub> queries into data stream queries in the target language SNEEqL in terms of the underlying stream schemas, and then executing these queries and returning the results as SPARQL bound variables. This component is called Semantic Integrator, whose core is implemented following the query rewriting principles described in Chapter 5. The other component is a service layer on top of the Semantic Integrator: the Integration and Query Service (IQS), which provides a web service interface according to the SemSorGrid4Env architecture specification.

<sup>1</sup>Code: <https://code.google.com/p/semanticstreams/>

### 7.1.1 Semantic Integrator

The integrator receives queries specified in terms of the classes and properties of the ontology using SPARQL<sub>Stream</sub>. In order to transform the SPARQL<sub>Stream</sub> query, expressed in terms of the ontology, into queries in terms of the data sources, a set of mappings must be specified in R2RML. This target of the *query rewriting* process is the continuous query language SNEEqL, which is expressive enough to deal with both streaming and stored sources. Note that query execution in sources such as sensor networks may include in-network query processing, pull or push based delivery of data between sources, and other data source specific settings. The result of the query processing is a set of tuples that the *data translation* process transforms into ontology instances.

**Query Rewriting:** During query rewriting, the following phases can be identified:

- **Query Parsing.** First the incoming SPARQL<sub>Stream</sub> query, specified in terms of an ontological schema, is parsed using an extended version of the ARQ SPARQL parser<sup>1</sup>. Consequently the result of this phase is an extension of the ARQ SPARQL abstract syntax tree. This tree can be easily explored to find the elements that will later be mapped and translated.
- **Mapping Reading.** The stream-to-ontology mapping is read and the relevant mappings to the SPARQL<sub>Stream</sub> input query are extracted. These mappings are provided as an input to the translation phase. The handling of the R2RML mapping language is provided by the Morph RDB2RDF tool<sup>2</sup>.
- **Rewriting.** The rewriting phase uses the SPARQL<sub>Stream</sub> syntax tree and the relevant mapping definitions to construct streaming relational algebra trees. These algebra expressions can be optimized using standard static optimization techniques, as in Chapter 5.

**Query Processing.** In this stage the Semantic Integrator delegates query execution to the distributed query processing service (Calbimonte et al., 2011a). In order to be able to be used by the Query Executor sub-component, a wrapper class must be implemented for the specific engine. Currently it is a wrapper of a SemSorGrid4Env web-service implementation of the streaming data service. Alternatively, the SNEE engine is supported natively through its Java API.

---

<sup>1</sup><http://jena.sourceforge.net/ARQ/>

<sup>2</sup>Morph: <https://github.com/jpcik/morph>

**Data Translation.** After the results are returned by the Query Processing component, they are translated again to instances of the ontologies referenced in the original query. This process requires again the mappings of the translation phase and results are returned as SPARQL bound variables.

### 7.1.2 Integration and Query Service

The functionality exposed by the IQS is specified by the Integration, Query and Pull Data interfaces of the SemSorGrid4Env architecture (Gray et al., 2009). The available methods for the integration interface are:

- **IntegrateAs:** Creates a data resource which presents the global view over a set of data sources. Takes as an input a data resource name (e.g. a sensor data source) and a mapping document (i.e. an R2RML file). The result is a new data resource to which SPARQL<sub>Stream</sub> queries can be posed.
- **AddSource:** Add one or more data sources to the set of known data sources. Takes as input the names of the resources to add.
- **RemoveSource:** Remove one or more data sources from the set of known data sources. Takes as input the names of the resources to remove.

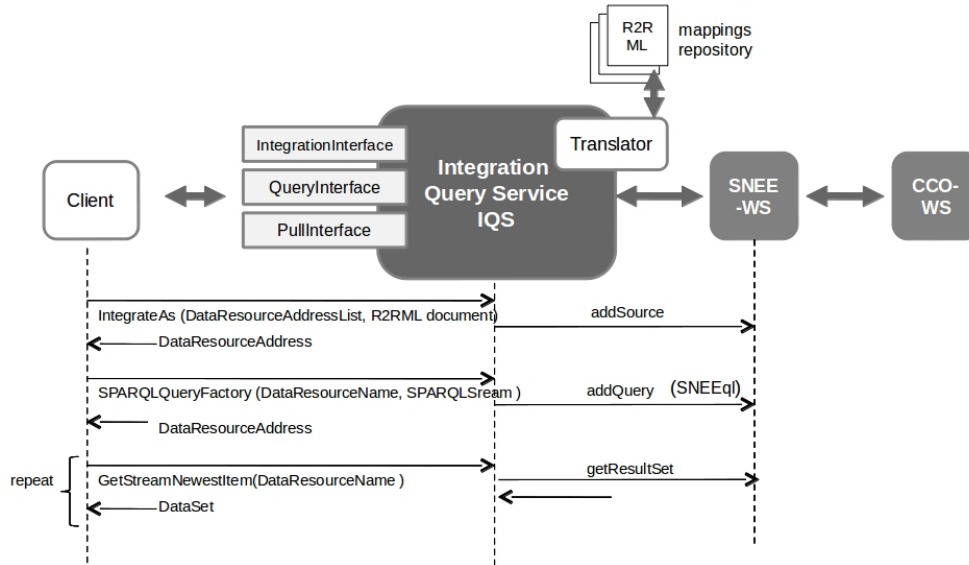
The operations for the Query Interface:

- **GetSPARQLPropertyDocument:** Returns the core property document values associated with the service implementing this message. The property document contains semantically annotated metadata about the query resource.
- **DestroyDataResource:** Destroy the named data resource; future messages directed at the resource must yield a fault indicating an invalid resource. Therefore the data resource is no longer available for querying.
- **SPARQLExecute:** Directs a query document to a data resource. Takes as an input the name of the resource to be queried (e.g. a SPARQL<sub>Stream</sub> service endpoint) and the query to be executed. This operation launches a one-off query, and returns the results immediately.
- **SPARQLExecuteFactory:** Creates a relationship between a data resource representing the result of a query and the data access service by which it will be accessed. This operation will create a continuous query (e.g. a SPARQL<sub>Stream</sub> query) whose

results will be accessible in a data resource accessible for pull access. The operation returns the name of the data resource associated to the continuous results.

Finally the operations of the data interface:

- **GetDataResourcePropertyDocument:**Retrieves the PropertyDocument for the pull stream response resource.
- **GetStreamItem:** Retrieves a specified count of stream items from the pull stream service from a specific point in the available history. Takes as input the data resource name, data format, number of maximum results and an offset.
- **GetStreamNewestItem:**Retrieves the most recent stream items in the pull stream response resource up to the specified count.



**Figure 7.2:** IQS Integration and Query Service operations interactions.

The service implementation is based on the capabilities of the Semantic Integrator library described in Section 7.1.1. Examples and details of the whole process of query translation and execution is provided in Chapter 4 and in (Calbimonte et al., 2011a).

### 7.1.3 Use-case: Coastal Sensors for Flood Warning

One of the use cases of SemSorGrid4Env includes a flood response planning application that uses data from sensor networks continuously monitoring the sea-state in the south-

ern coast of England. For this application, the sensor readings need to be put in context by integrating them with other sources of data about the surrounding environment.

A key challenge in this context is integrating data from heterogeneous sources, both in terms of the data modality (i.e. streaming and stored) and its representation (i.e. the schema used). We addressed this problem by using the IQS service and the Semantic Integrator described previously (Section 7.1.1). Through the Integration interface, the semantic integrator supports the creation of a virtual data source in which the data from multiple physical data sources appear to co-exist in a single data model. The virtual data source can be queried through the Query interface and query answers retrieved through either the Data Access or Subscription interfaces, as described in Section 7.1.2.

One example SPARQL<sub>Stream</sub> query is presented in Listing 7.1. It characterizes an over-topping event over an ontological observation model based on the SSN Ontology<sup>1</sup> for sea-state readings. The event is characterized by the measured wave height being greater than the associated storm threshold value for a specific sensor.

---

```
PREFIX ssn: <http://purl.oclc.org/NET/ssnx/ssn#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX CoastalDefences: <http://www.semsorgrid4env.eu/ontologies/CoastalDefences.owl#>

SELECT ?location ?waveHeight ?stormThreshold ?observedTime
FROM NAMED STREAM <http://semsorgrid4env.eu/ns#waveStream.srdf>[NOW]
WHERE {
    ?WaveObs rdf:type ssn:Observation;
              ssn:observationResult ?waveHeight;
              ssn:observationResultTime ?observedTime;
              ssn:observedProperty ?waveProperty;
              ssn:featureOfInterest ?defence.
    ?defence CoastalDefences:hasLocation ?location;
              CoastalDefences:hasStormThreshold ?stormThreshold.
    ?waveProperty rdf:type CoastalDefences:WaveHeight.
    FILTER(?waveHeight > ?stormThreshold).
}
```

---

**Listing 7.1:** SPARQL<sub>Stream</sub> query that characterizes an over-topping event in terms of an observation ontology.

---

<sup>1</sup>SemSorGrid4Env ontologies: <http://www.semsorgrid4env.eu/ontologies/>

### 7.1.4 Rewriting Examples for Coastal Sea Sensors

Most of the real SemSorGrid4Env data sources are low throughput environmental sensor streams, with update rates that range around 5 or 10 minutes. We have worked with a series of streams, which can be classified in three groups: meteorological, wave streams and tide streams. For each sensor location, a different extent name exists. So for instance, a tide stream for a deployment in Hernebay is named `envdata_hernebay_tide`. We provide the stream schema of a meteorological sensor in Listing 7.2.

---

```
envdata_folkestone: {  
    Timestamp integer,  
    DateTime string,  
    TAir float,  
    WDir float,  
    GustSpeed float,  
    WindSpeed float,  
    AirPressure float }
```

---

**Listing 7.2:** Example of a meteorological stream schema.

The `Timestamp` attribute represents the stream tuple time, used for stream ordering and windowing. The rest of the attributes contain the observation values. For instance `TAir` represents the measured air temperature value, or the `WindSpeed` represents the wind-speed value measured in that location (i.e. the Folkestone meteorological sensor station). We detail some example queries in `SPARQLStream` and its corresponding rewritten query in `SNEEql`.

**Time-based window query** This simple `SPARQLStream` query (Listing 7.3) requests the wave height observed values (i.e. observations with observed property `cd:WaveHeight`) of the last 10 minutes, observed by the Milford station sensor.

---

```
PREFIX sb: <http://www.w3.org/2009/SSN-XG/Ontologies/SensorBasis.owl#>  
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>  
PREFIX ssn: <http://purl.oclc.org/NET/ssnx/ssn#>  
PREFIX cd: <http://www.sensorsgrid4env.eu/ontologies/CoastalDefences.owl#>  
PREFIX ssg: <http://sensorsgrid4env.eu/ns#>  
  
SELECT ?wavets ?waveheight  
  
FROM NAMED STREAM <http://sensorsgrid4env.eu/ns#ccometeo.srdf>  
  
[NOW - 10 MINUTE SLIDE 10 S]  
  
WHERE {  
    ?WaveObs a ssn:Observation;  
        ssn:observationResultTime ?wavets;  
        ssn:observationResult ?waveheight;
```

```

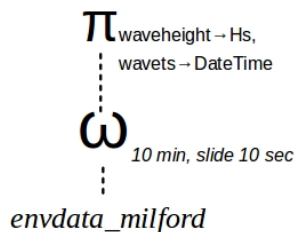
    ssn:observedProperty cd:WaveHeight;
    ssn:observedBy ssg:MilfordSensor.
}

```

---

**Listing 7.3:** SPARQL<sub>Stream</sub> query requesting wave height observations at Milford in the latest 10 minutes.

In this case, there is no need to request any stream other than the one that holds the Milford data: *envdata\_milford*. Also, a sliding window of 10 minutes is applied to this stream, as shown in the rewritten query algebra expression in Figure 7.3 (and the corresponding SNEEql query in Listing 7.4), using the semantics of Chapter 5.



**Figure 7.3:** Algebra expression generated for the SPARQL<sub>Stream</sub> query in Listing 7.3.

---

```

(SELECT Hs AS waveheight, DateTime AS wavets
FROM envdata_milford[FROM NOW-10 MINUTES TO NOW-0 MINUTES SLIDE 10 SECONDS] envdata_milford);

```

---

**Listing 7.4:** SNEEql query rewritten.

**Union queries** In this example, instead of focusing on a specific sensor location, we request all wave height observations available (Listing 7.5). Because there are several streams that provide wave height data, the resulting rewritten query performs a union over these streams, as seen in the algebra expression of Figure 7.4 (See the instantiated SNEEql query in Listing 7.6).

---

```

PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX ssn: <http://purl.oclc.org/NET/ssnx/ssn#>
PREFIX cd: <http://www.semsorgrid4env.eu/ontologies/CoastalDefences.owl#>
PREFIX ssg: <http://semsorgrid4env.eu/ns#>

SELECT ?wavets ?waveheight
WHERE {
    ?WaveObs a ssn:Observation;
    ssn:observationResultTime ?wavets;
    ssn:observationResult ?waveheight;

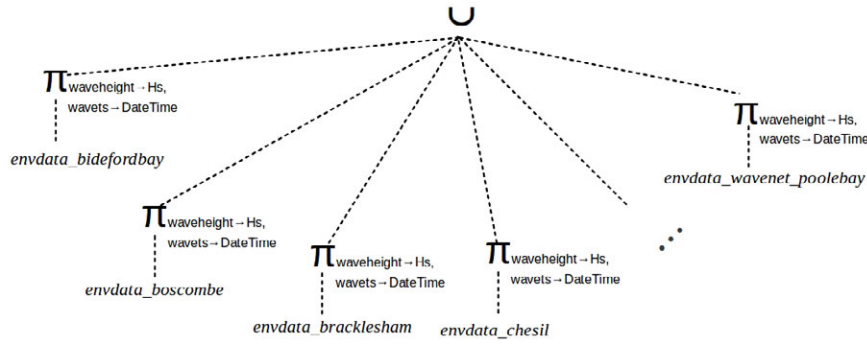
```

```

    ssn:observedProperty cd:WaveHeight;
    ssn:observedBy ?sensor.
}

```

**Listing 7.5:** SPARQL<sub>Stream</sub> query that requests wave height observations.



**Figure 7.4:** Algebra expression generated for the SPARQL<sub>Stream</sub> query in Listing 7.5.

```

(SELECT Hs AS waveheight, DateTime AS wavets FROM envdata_bidefordbay) UNION
(SELECT Hs AS waveheight, DateTime AS wavets FROM envdata_boscombe) UNION
(SELECT Hs AS waveheight, DateTime AS wavets FROM envdata_bracklesham) UNION
(SELECT Hs AS waveheight, DateTime AS wavets FROM envdata_chesil) UNION
(SELECT Hs AS waveheight, DateTime AS wavets FROM envdata_folkestone) UNION
(SELECT Hs AS waveheight, DateTime AS wavets FROM envdata_goodwin) UNION
(SELECT Hs AS waveheight, DateTime AS wavets FROM envdata_haylingisland) UNION
(SELECT Hs AS waveheight, DateTime AS wavets FROM envdata_hornsea) UNION
(SELECT Hs AS waveheight, DateTime AS wavets FROM envdata_looebay) UNION
(SELECT Hs AS waveheight, DateTime AS wavets FROM envdata_milford) UNION
(SELECT Hs AS waveheight, DateTime AS wavets FROM envdata_minehead) UNION
(SELECT Hs AS waveheight, DateTime AS wavets FROM envdata_penzance) UNION
(SELECT Hs AS waveheight, DateTime AS wavets FROM envdata_perranporth) UNION
(SELECT Hs AS waveheight, DateTime AS wavets FROM envdata_pevenseybay) UNION
(SELECT Hs AS waveheight, DateTime AS wavets FROM envdata_rhylflats) UNION
(SELECT Hs AS waveheight, DateTime AS wavets FROM envdata_rustington) UNION
(SELECT Hs AS waveheight, DateTime AS wavets FROM envdata_rye) UNION
(SELECT Hs AS waveheight, DateTime AS wavets FROM envdata_sandownbay) UNION
(SELECT Hs AS waveheight, DateTime AS wavets FROM envdata_seaford) UNION
(SELECT Hs AS waveheight, DateTime AS wavets FROM envdata_startbay) UNION
(SELECT Hs AS waveheight, DateTime AS wavets FROM envdata_torbay) UNION
(SELECT Hs AS waveheight, DateTime AS wavets FROM envdata_westbay) UNION
(SELECT Hs AS waveheight, DateTime AS wavets FROM envdata_westonbay) UNION

```



```
(SELECT Hs AS waveheight, DateTime AS wavets FROM envdata_weymouth) UNION
(SELECT Hs AS waveheight, DateTime AS wavets FROM envdata_wavenet_poolebay) ;
```

---

**Listing 7.6:** SNEEqL query rewritten performing a union over the streams.

**Stream join queries** A useful feature for environmental coastal sensor applications is the ability to correlate and combine data from different sensors. The following SPARQL<sub>Stream</sub> query (Listing 7.7) requests wave heights higher than a tide height in the Milford location.

---

```
PREFIX sb: <http://www.w3.org/2009/SSN-XG/Ontologies/SensorBasis.owl#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX ssn: <http://purl.oclc.org/NET/ssnx/ssn#>
PREFIX cd: <http://www.sensorgrid4env.eu/ontologies/CoastalDefences.owl#>
PREFIX ssg: <http://sensorgrid4env.eu/ns#>
SELECT ?wavets ?waveheight ?tideheight
WHERE {
    ?WaveObs a ssn:Observation;
        ssn:observationResultTime ?wavets;
        ssn:observationResult ?waveheight;
        ssn:observedProperty cd:WaveHeight;
        ssn:observedBy ssg:MilfordSensor.
    ?TideObs a ssn:Observation;
        ssn:observationResultTime ?tidets;
        ssn:observationResult ?tideheight;
        ssn:observedProperty cd:TideHeight.
    FILTER (?waveheight>?tideheight)
}
```

---

**Listing 7.7:** SPARQL<sub>Stream</sub> query that joins wave heights in Milford with tide heights if the first are greater.

The rewritten query (Listing 7.8) joins data from meteorological sensors and tide sensors, and also performs a union because several different streams provide tide information.

---

```
(SELECT envdata_milford.Hs AS waveheight, envdata_milford.DateTime AS wavets,
    envdata_deal_tide.Tp AS tideheight
FROM envdata_milford, envdata_deal_tide WHERE Hs > Tp) UNION
(SELECT envdata_milford.Hs AS waveheight, envdata_milford.DateTime AS wavets,
    envdata_hernebay_tide.Tp AS tideheight
FROM envdata_milford, envdata_hernebay_tide WHERE Hs > Tp) UNION
```

## 7.2. Swiss Experiment Metadata and Querying

---

```
(SELECT envdata_milford.Hs AS waveheight, envdata_milford.DateTime AS wavets,  
      envdata_lymington_tide.Tp AS tideheight  
FROM envdata_milford, envdata_lymington_tide WHERE Hs > Tp) UNION  
(SELECT envdata_milford.Hs AS waveheight, envdata_milford.DateTime AS wavets,  
      envdata_sandownpier_tide.Tp AS tideheight  
FROM envdata_milford, envdata_sandownpier_tide WHERE Hs > Tp) UNION  
(SELECT envdata_milford.Hs AS waveheight, envdata_milford.DateTime AS wavets,  
      envdata_swanagepier_tide.Tp AS tideheight  
FROM envdata_milford, envdata_swanagepier_tide WHERE Hs > Tp) UNION  
(SELECT envdata_milford.Hs AS waveheight, envdata_milford.DateTime AS wavets,  
      envdata_teignmouthpier_tide.Tp AS tideheight  
FROM envdata_milford, envdata_teignmouthpier_tide WHERE Hs > Tp) UNION  
(SELECT envdata_milford.Hs AS waveheight, envdata_milford.DateTime AS wavets,  
      envdata_westbaypier_tide.Tp AS tideheight  
FROM envdata_milford, envdata_westbaypier_tide WHERE Hs > Tp) ;
```

---

**Listing 7.8:** SNEEql query rewritten joining a meteorological stream and tide streams.

## 7.2 Swiss Experiment Metadata and Querying

In this section, we show how we used SPARQL<sub>Stream</sub> and our query rewriting approach for a federated sensor network environment in the Swiss-Experiment project. Swiss-Experiment is a collaborative platform for sharing real-time sensor data across various institutions to improve environmental hazard forecasting and warning. In this platform, the sensor data is maintained with Global Sensor Networks (GSN) ([Aberer et al., 2006](#)), a middleware that supports flexible integration of sensor networks and sensor data, provides distributed querying and filtering, as well as dynamic adaptation and configuration. The Swiss-Experiment project has several GSN instances deployed in different locations which operate independently. In this way they can efficiently perform their query operations locally, and can be accessed using RESTful service interfaces, as mentioned earlier in [Section 5.7](#).

Using semantically rich models to model sensor data, is useful for interconnecting, sharing, reusing and linking data with other sources. We show in [Section 7.2.1](#) the process of building such a model, based on the SSN ontology, and then in [Section 7.2.2](#) we provide details on how we used SPARQL<sub>Stream</sub> to write queries based on these ontologies, while the data was managed by GSN.

### 7.2.1 Modeling Sensor Data with the SSN Ontology

In a highly heterogeneous setting such as the Swiss-Experiment, using standards and widely adopted vocabularies facilitates the tasks of publishing, searching and sharing sensor data. As seen in Section 2.4.2, the W3C SSN XG group introduced a generic and domain independent model, the SSN ontology, capable of representing sensor metadata and observations.

In Swiss-Experiment, sensors of different characteristics are deployed on different experiment sites, mainly in remote locations of the Swiss Alps. These field sites group different sensing platforms, each of which includes one or more sensing devices. For example, Wannengrat is a high altitude alpine fieldsite, centred around a single catchment ranging in altitude from 1500 to 2700m. The site has 7 semi-permanent meteo stations with additional mobile stations deployed temporarily for specific studies.

In Wannengrat, each of the meteo stations includes several sensor devices, for instance in station *wan7* an AlpuG IR sensor measures the snow surface temperature, and a Young 05103 Wind Monitor device measures the wind speed and direction. Although both measure very different properties, they are both observations in terms of the SSN ontology. Each of these observations observe a different property (`ssn:observedProperty`), of a given feature of interest (`ssn:featureOfInterest`), in this case the snow surface or the wind, in that location. Each observation is performed by a different sensor (`observedBy` property), and the wind sensor is capable observing more than one property (e.g. wind speed and wind direction are two distinct properties).

The SSN Ontology does not include specific classes or instances for these features or properties, because it is a general purpose model, intended to be extended or complemented with domain-specific ontologies. Following this approach, we have used the Climate & Forecast ontologies (`cf-feature`<sup>1</sup> and `cf-property`<sup>2</sup>), based on the vocabulary maintained by the Program for Climate Model Diagnosis and Intercomparison<sup>3</sup>. The vocabulary for Quantity Kinds and Units (`qu`<sup>4</sup>), based on the QUDV model<sup>5</sup>. Other ontologies for space and time specification can also be used in conjunction with SSN, such

---

<sup>1</sup>Climate & Forecast features: [www.w3.org/2005/Incubator/ssn/ssnx/cf/cf-feature.html](http://www.w3.org/2005/Incubator/ssn/ssnx/cf/cf-feature.html)

<sup>2</sup>Climate & Forecast properties: [www.w3.org/2005/Incubator/ssn/ssnx/cf/cf-property.html](http://www.w3.org/2005/Incubator/ssn/ssnx/cf/cf-property.html)

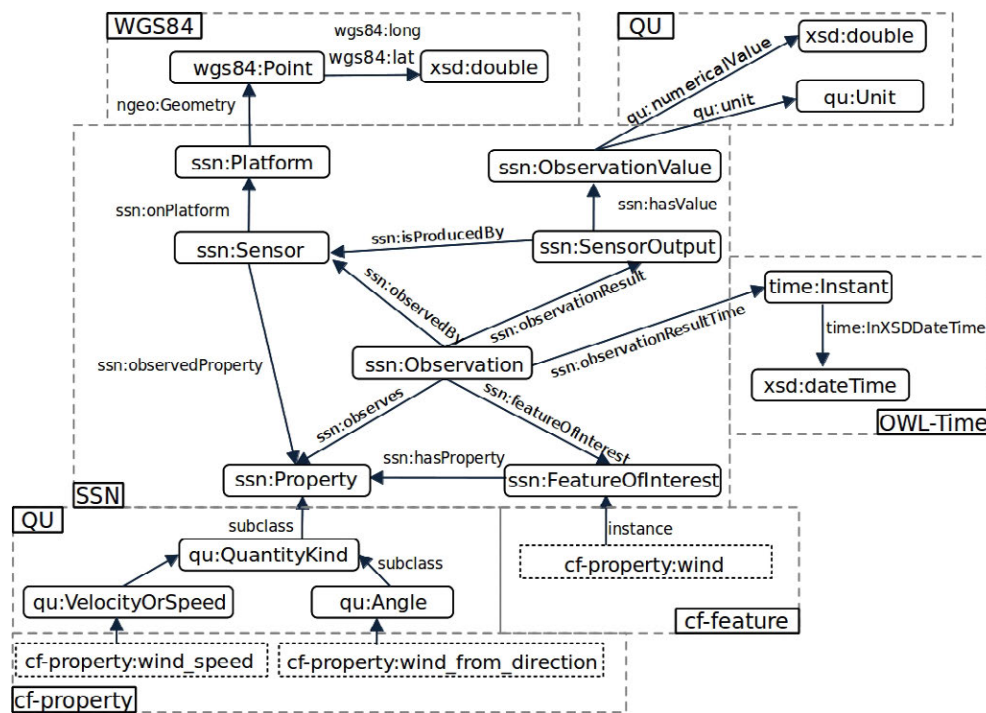
<sup>3</sup>PCMDI: <http://www-pcmdi.llnl.gov/>

<sup>4</sup>Library for Quantity Kinds and Units: <http://purl.oclc.org/NET/ssnx/qu/qu>

<sup>5</sup>QUDV: <http://www.omgsysml.org/qudv>

as WGS84<sup>1</sup>, GeoVocab<sup>2</sup> and the Time Ontology<sup>3</sup>.

We can see some of the relationships between SSN and the other ontologies in Figure 7.5. Notice that concrete observed properties are defined in the cf-property ontology in this example, such as cf-property:wind\_speed, but we could further extend the ontology network with other vocabularies if we need additional concepts. Also notice that these concrete properties can be classified, as it is the case in the example of wind\_speed, as a sub type of VelocityOrSpeed, which is a subtype of QuantityKind.



**Figure 7.5:** SSN Ontology combined with other vocabularies for representing sensor observations.

In Listing 7.9 we show an example of a wind speed observation using the SSN Ontology network.

```

swissex:WindSpeedObservation1 rdf:type ssn:Observation ;
    ssn:observedBy swissex:SensorWindWan7;
    ssn:featureOfInterest cf-feature:wind;
    ssn:observedProperty cf-property:wind_speed;
    ssn:observationResultTime [time:inXSDDateTime "2011-08-26T21:32:52"^^xsd:dateTime];
    ssn:observationResult [rdf:type ssn:SensorOutput ;
        ssn:hasValue swissex:WindSpeedObservationValue1 ].

```

<sup>1</sup>WGS84:<http://www.w3.org/2003/01/geo/>

<sup>2</sup>GeoVocab: <http://geovocab.org>

<sup>3</sup>OWL-Time:<http://www.w3.org/TR/owl-time/>

```

swissex:WindSpeedOvervationValue1
  qudt:numericValue "6.245"^^ xsd:double;
  qu:unit unit:metrePerSecond.

```

---

**Listing 7.9:** Wind speed observation in terms of the SSN Ontology.

In this representation, we can identify the main attributes of an observation, including the sensor that performed the observation (`swissex:SensorWindWan7`), the feature of interest (i.e. `cf-feature:wind`), the observed property, which is wind speed, the observation time instant, and the observation value, including the numerical data and the unit of measurement. Actual values of the sensor output can be represented as instances linked to the `ssn:SensorOutput` class through the `hasValue` property. The data itself can be linked through a specialized property of a quantity ontology (e.g. the `qu:numericValue` property). This model allows representing other types of observation in a very similar fashion. For instance, for a wind direction observation, the structure remains the same, but the observed property would change to `wind_from_direction` and the unit of measurement would have to be changed accordingly.

In this example we used the basic SSN Ontology classes and properties. However it could be desirable to extend those to specific ones. For instance instead of using `ssn:Observation`, we could create a subtype `swissex:WindSpeedObservation` whose feature of interest is always wind, and observed property is fixed to wind speed.

### 7.2.2 Querying GSN with SPARQL<sub>Stream</sub>

With this model available, users are expected to write queries in terms of observations, features of interest and observed properties, and we describe how this was done using SPARQL<sub>Stream</sub> in this section. The GSN middleware used in Swiss-Experiment internally uses an in-memory data management system, and allows defining continuous queries through configuration and accessing virtual streams through a RESTful service interface.

**Virtual Sensors** The GSN virtual sensors in Swiss-Experiment are given a name and a set of attributes, each with a data type (e.g. integer, double, etc.). These attributes hold the observed values by the sensors. In general, Swiss-Experiment virtual sensors group observed values from physical sensors of a given station. That is why a virtual sensor such as `wannengrat_wan7` may have attributes ranging from wind speed to snow height or radiation, as seen in Figure 7.6. Each virtual sensor also exposes a timed attribute which indicates the time the observation was registered. Because these observations are

aggregated and collected from different devices, this timestamp is not necessarily the actual observation time but for simplicity can be assumed to be so.

wannengrat_wan7 16/07/2012 08:30:00 +0100			
Real-Time	Addressing	Structure	Description
		record	double
		relative_humidity	double
		air_temperature	double
		wind_speed_scalar_av	double
		wind_speed_vector_av	double
		wind_direction	double
		wind_direction_stdev	double
		wind_speed_max	double
		snow_surface_temperature	double
		snow_height	double
		incoming_shortwave_radiation	double
		outgoing_shortwave_radiation	double
		incoming_longwave_radiation	double
		outgoing_longwave_radiation	double

**Figure 7.6:** Swiss Experiment virtual sensor wannengrat\_wan7.

**Mappings** R2RML mappings are used to relate the virtual sensors to the ontological model described in Section 7.2.1. The simple example in Figure 7.7 shows an example of a mapping of the wannengrat\_wan7 sensor to a relative humidity `ssn:ObservationValue`. The sensor name is given as the `rr:tableName` in the mapping definition. For each tuple of this virtual sensor, a triple of type `ssn:ObservationValue` will be created, whose URI is generated through a `rr:template` string. This is way of concatenating arbitrary strings with an attributed of the sensor, in this case the timed timestamp. Notice that the URI of this observation value ensures uniqueness, by prefixing the sensor name and observed property to the timestamp. Another triple is specified with the `rr:predicateObjectMap`, with the same subject, a fixed predicate (`qu:numericalValue`) and an object given by the `relative_humidity` column. Mappings for the observation itself, the sensor and other elements described before, have been created in the same way.

**Queries** With the provided mappings, we can write a simple SPARQL<sub>Stream</sub> query that returns the latest 10 minutes of relative humidity in wan7 (Listing 7.10).

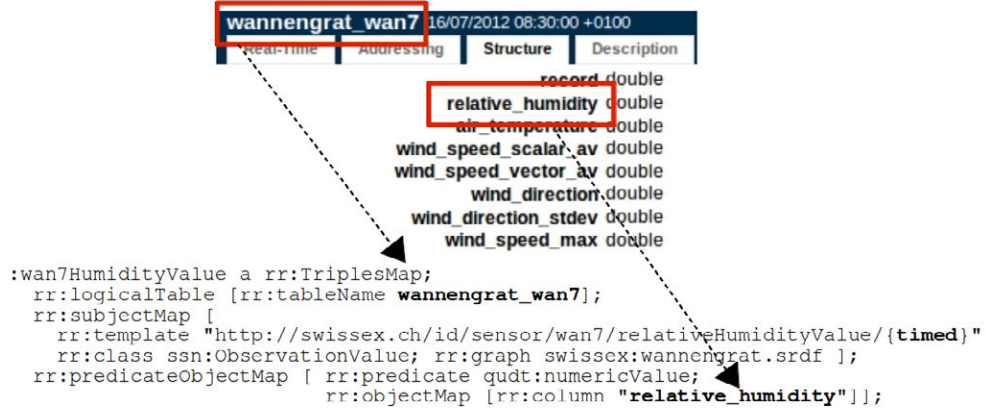
---

```

PREFIX ssn: <http://purl.oclc.org/NET/ssnx/ssn#>
PREFIX qu:  <http://purl.oclc.org/NET/ssnx/qu#>

SELECT ?obs ?val
FROM NAMED STREAM <http://swiss-experiment.ch/data#wannengrat.srdf>[NOW-10 MINUTES]
WHERE {
  ?obs a ssn:Observation;
    ssn:observedProperty cf-property:relative_humidity;
    ssn:observedBy swissex:Wan7Sensor;

```



**Figure 7.7:** Virtual sensor wannengrat\_wan7 mapped to an Observation Value in the SSN Ontology.

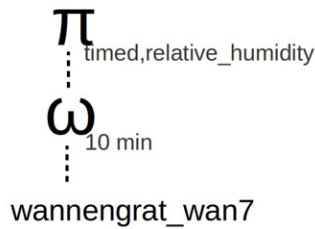
```

    ssn:observationResult ?output.
  ?output ssn:hasValue ?obsVal.
  ?obsVal qudt:numericValue ?val.
}

```

**Listing 7.10:** Query the relative humidity observations in the latest 10 minutes in wan7 station.

This simple query can be rewritten using the approach in Chapter 5, to the following expression in Figure 7.8. It simply projects the attributes from a 10-minute window applied to the wannengrat\_wan7 virtual sensor.



**Figure 7.8:** Rewritten algebra expression in terms of the wannengrat\_wan7 virtual sensor.

This is trivially transformed to a RESTful service request to the GSN server, as in Listing 7.11. The virtual sensor is specified as the `vs` parameter and the attribute list are the `field` parameter. The window is denoted with the `from-to` parameter combination.

```

http://montblanc.slf.ch:22001/multidata?
  vs[0]=wannengrat_wan7&
  field[0]=relative_humidity&
  from=15/08/2012+05:00:00&to=15/08/2012+15:00:00

```

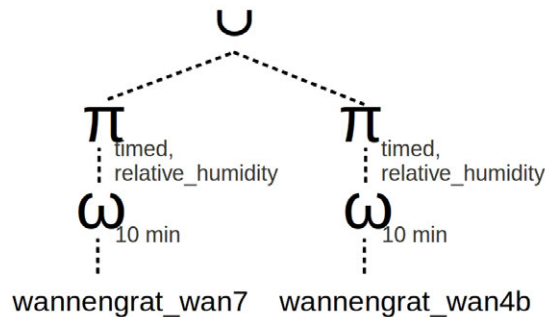
**Listing 7.11:** Query the relative humidity observations in the latest 10 minutes in wan7 station.

In this case the query is launched on a single virtual sensor but consider a very similar one (Listing 7.12), that does not ask for data from a particular station (i.e. omitting the `ssn:observedBy` triple pattern).

```
PREFIX ssn: <http://purl.oclc.org/NET/ssnx/ssn#>
PREFIX qu:  <http://purl.oclc.org/NET/ssnx/qu#>
SELECT ?obs ?val
FROM NAMED STREAM <http://swiss-experiment.ch/data#wannengrat.srdf>[NOW-10 MINUTES]
WHERE {
  ?obs a ssn:Observation;
      ssn:observedProperty cf-property:relative_humidity;
      ssn:observationResult ?output.
  ?output ssn:hasValue ?obsVal.
  ?obsVal qu:numericalValue ?val.
}
```

**Listing 7.12:** Query the relative humidity observations in the latest 10 minutes.

This simple query does not indicate a specific station so it is implied that it will retrieve all relative humidity observations from the data source. In terms of query rewriting in GSN, if we have two virtual sensors mapped to relative humidity, (for instance `wannengrat_wan7` and `wannengrat_wan4b`, then the query rewriting would need to get data from both in GSN and then perform a union, depicted in Figure 7.9.



**Figure 7.9:** Rewritten algebra expression as a union of the `wannengrat_wan4b` and `wannengrat_wan7` virtual sensors.



### 7.3 Ciudad2020 Bike Sharing

Bicycle sharing systems in different cities publish their data on the Web. These bike sharing services distribute bike stations in different key points in the city and a user may pick up a bike in one station and drop it off at another one. In the context of the Ciudad2020 project<sup>1</sup>, aiming at open city data access, we used bike sharing as a case study for live-publishing of bikes availability in different systems.

Currently there is an API<sup>2</sup> that gathers bike systems data and exposes it either in JSON or in HTML format. We used this API to generate both static data about the bike systems (including station geographical locations, related places of interest, etc.) and dynamic data using virtual RDF streams accessible through SPARQL<sub>Stream</sub>.

The static data was generated using a classical RDB2RDF generation approach (Ruckhaus et al., 2012) and is out of the scope of this thesis. However, it is interesting to notice that the mappings and modeling used for the static data are reused for with the dynamic data.

#### 7.3.1 Bike Sharing Modeling

This ontology model is based on the data source schemas and aims at reusing existing ontologies and vocabularies. The data is originally provided in JSON format, structured in simple name-value paired tuples. For a given bike system (typically one per city), the list of station and the bike availability is given by a JSON dataset as the example in Listing 7.13.

---

```
{ "id" : "0", "name": "001 - Ro Balsas-Ro Sena",
  "lat": "19433296", "lng": "-99168051",
  "timestamp": "2012-03-09 14:19:28.420714",
  "bikes": 16, "free": 11 },
{ "id": "1", "name": "002 - Ro Guadalquivir - Ro Balsas",
  "lat": "19431386", "lng": "-99171695",
  "timestamp": "2012-03-09 14:19:31.302252",
  "bikes": 6, "free": 6 }
```

---

**Listing 7.13:** Sample JSON data from the citibik.es API.

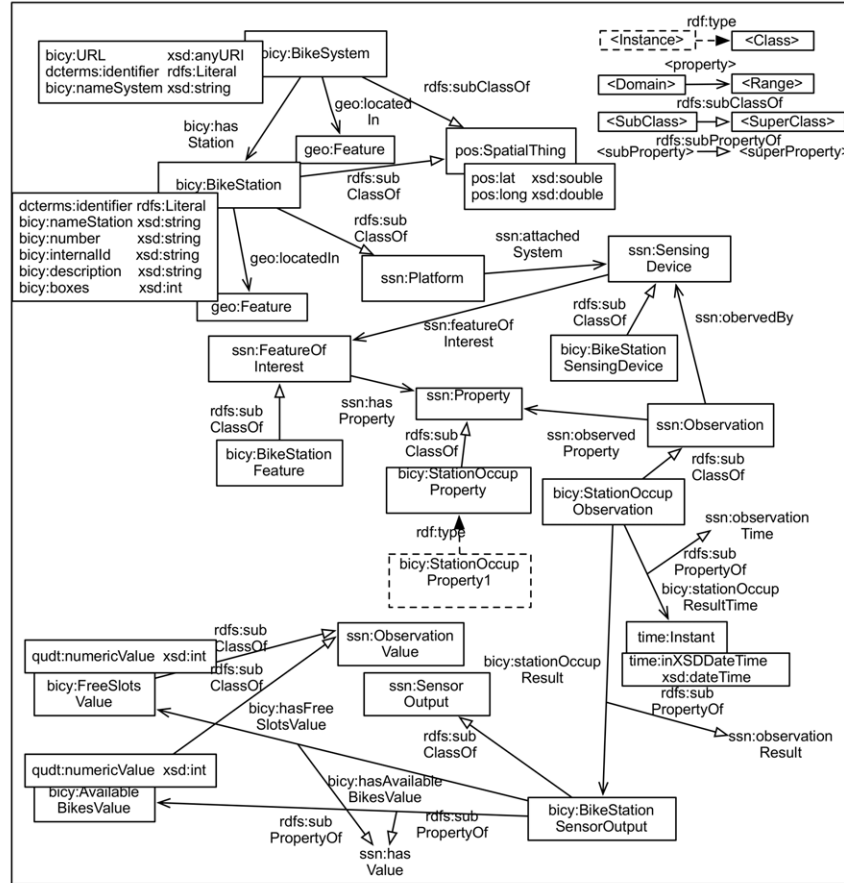
We have built a citybikes ontology network<sup>3</sup>, depicted in Figure 7.10, which represents observations of available bikes and free slots in every station of the participating by bike sharing systems in different cities.

---

<sup>1</sup>Proyecto Innpronta Ciudad2020: [www.innprontaciudad2020.es/](http://www.innprontaciudad2020.es/)

<sup>2</sup>Citybik.es API: <http://api.citybik.es/>

<sup>3</sup><http://transporte.linkeddata.es/models.html>



**Figure 7.10:** SSN based Bike Sharing Ontology

Each of these measurements represents the state of a bike station in a particular place and time, and is conducted through a sensor in each bike station. The citybikes ontology network follows a modular structure consisting of a central ontology that is related to a set of ontologies that describe different sub domains involved in the modeling of the bike station measurements. The central ontology is the bike sharing system ontology which contains concepts such as the bike sharing system and its name, the bike station and its name, number, internal id, status, description, number of boxes, free slots and free bikes. We have selected the following resources:

- The Dublin Core terms ontology (<http://purl.org/dc/terms/>) for identifiers.
- The Semantic Sensor Network ontology (<http://purl.oclc.org/NET/ssnx/ssn#>) for sensors and observations. In this context, there is a sensor in each bike station, and the observations are the number of available bikes and free slots at a certain date

and time.

- The positioning ontology ([http://www.w3.org/2003/01/geo/wgs84\\_pos](http://www.w3.org/2003/01/geo/wgs84_pos)) for geo positioning the systems and stations with latitude and longitude.
- The time ontology (<http://www.w3.org/2006/time#>) to represent the timestamp as an instant.
- The Geonames ontology (<http://www.geonames.org/ontology#>), used to define the location of the bike sharing systems and the stations based on their latitude and longitude.
- The quantities, units and dimensions ontology (<http://qudt.org/>) for the number of available bikes and free slots.

Once the ontology has been defined, a resource naming strategy has to be defined in order to ensure that every class in the ontology can have individuals with unique identifiers (i.e., URIs).

The ontology network is focused on defining subclasses and subproperties of the SSN ontology. In our use case, the concepts bike station, platform and feature of interest have a one to one relationship and could be merged as one only concept, i.e., Bike Station. However, to comply with the SSN ontology, the three concepts were kept.

### 7.3.2 Querying Bike Observations

The generation of RDF from the selected data sources requires first, to transform the data from the data sources into RDF and to evaluate the generated data to verify that the transformation was correct. Second, the dataset has to be linked to other relevant datasets by identifying and validating links between resources in the generated dataset and external ones. And, third, the description of the generated dataset must be created (e.g., using the VoID vocabulary<sup>1</sup>), including the relevant metadata about it.

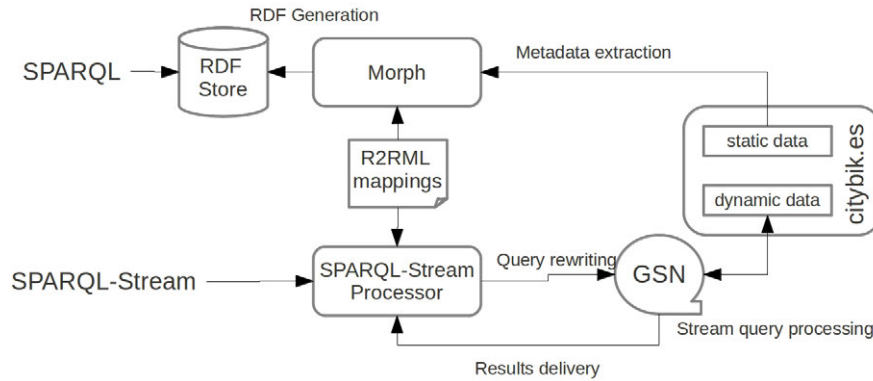
In our use case, we required publishing and consuming two types of data: stored and streaming data. The first type is related to the metadata and contextual information about sensor data, including geographical location, sensor and station characteristics, observed features, and also includes the data about points of interest (museums, libraries, city guides, etc.). The streaming data, in contrast, is centered on the observations of free slots and available bikes.

Therefore, we used an approach (depicted in Figure 7.11) that relies on ontology-based query rewriting of SPARQL<sub>Stream</sub> (Calbimonte et al., 2010b). In particular, we used

---

<sup>1</sup><http://www.w3.org/TR/void/>

GSN Aberer et al. (2006) (Global Sensor Networks), to which we added a wrapper to the citybik.es API service. SPARQL<sub>Stream</sub> queries are rewritten to GSN requests using the techniques mentioned in Chapter 5.



**Figure 7.11:** The implemented approach for generating static RDF and querying RDF streams with SPARQL<sub>Stream</sub> from the citybik.es services.

A key advantage of using SPARQL<sub>Stream</sub> is that it also uses R2RML mappings for streaming data queries (Calbimonte et al., 2012a). This feature allowed us to use the same set of mapping definitions for both the static and streaming data, even though we used Morph for the first one and SPARQL<sub>Stream</sub> for the second one. Notice that both approaches follow very different RDF management strategies: For static stored data we generate materialized RDF triples that can be later queried in a standard triple store. For dynamic data we pose queries to a virtual streaming RDF dataset, and the queries are rewritten by a SPARQL<sub>Stream</sub> processor to the underlying stream processing engine, which throws the query results.

### 7.3.3 Exploitation

In terms of exploitation (e.g. building applications on top of the generated dataset), we have made the data available in two different types of endpoints (one for static data and the another one for dynamic data). The complete description of the application is now available at <http://transporte.linkeddata.es/>. To facilitate the use of this data, we have provided two types of sample queries in the aforementioned website: queries against static data, and queries against dynamic data.

In particular, sample SPARQL<sub>Stream</sub> queries can be run in a demo web interface<sup>1</sup>. A sample query for this service is given below in Listing 7.14. It asks for the number of

<sup>1</sup><http://streams.linkeddata.es/query/citybikes>

available bikes and free slots in all stations in the last 5 minutes.

---

```

PREFIX ssn: <http://purl.oclc.org/NET/ssnx/ssn#>
PREFIX qudt: <http://data.nasa.gov/qudt/owl/qudt#>
PREFIX bicy: <http://citybikes.linkeddata.es/ontology#>
PREFIX time: <http://www.w3.org/2006/time#>
PREFIX dcterms: <http://purl.org/dc/terms/>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>

SELECT ?sensdev ?avbikes ?freeslots ?tstamp
FROM NAMED STREAM <http://citybikes.linkeddata.es/ontology#CityBikes.srdf> [NOW - 300 S]
WHERE {
    ?obs ssn:observedBy ?sensdev .
    ?obs a bicy:FreeBikesObservation .
    ?obs ssn:observationResult ?output .
    ?output bicy:hasAvailableBikesValue ?av .
    ?av qudt:numericValue ?avbikes .
    ?output bicy:hasFreeSlotsValue ?fs .
    ?fs qudt:numericValue ?freeslots .
    ?obs ssn:observationResultTime ?i .
    ?i time:inXSDDateTime ?tstamp
}
```

---

**Listing 7.14:** Query the number of available bikes and free slots in all stations in the last 5 minutes.

## 7.4 Discussion

In this chapter we have presented three use cases that show the applicability of the ontology-based streaming data access approach described in Chapters 4 and 5. Concretely, the use of the SPARQL<sub>Stream</sub> language and its query rewriting semantics have been shown to be applicable using different underlying query technologies, domains and scenarios, namely:

- In-network and out-of network query processing engines: SNEE in SemSorGrid4Env.
- Sensor data middleware: GSN, REST APIs in Swiss Experiment and Ciudad2020.

Besides, we have seen the applicability of R2RML mappings for exposing streaming data through SPARQL<sub>Stream</sub>. These mappings are in fact the same mappings defined for static RDF data generation, and follow the W3C Recommendation specification without

changes, because we based the  $\text{SPARQL}_{\text{Stream}}$  query rewriting semantics in terms of relational algebra expressions that fit naturally with the R2RML concepts.



## Chapter 8

# Evaluation

This chapter presents the evaluation carried out to validate the hypotheses stated in Chapter 3, addressing the corresponding research problems: **P1**: *How to access streaming sensor data with continuous queries using ontology models to represent the sensor observations.* and **P2**: *How to characterize sensor data using the recorded time series, capturing the semantic metadata of the sensor observations.*

We proceeded to validate each hypothesis with the evaluation results presented throughout this chapter. First we focus on the evaluation of the ontology-based query rewriting of SPARQL<sub>Stream</sub> in Section 8.1, related to hypotheses H3 and H5. The expressiveness of the SPARQL<sub>Stream</sub> query language compared to different underlying streaming query languages in Section 8.2, covering H2 and H4. The functional evaluation in Section 8.3 compares SPARQL<sub>Stream</sub> to other query languages for SPARQL streaming extensions, related also to H2. We show an evaluation of the sensor data characterization approach discussed in Chapter 6 in Section 8.4, which validates H6 and H7.

Finally we provide our conclusions about the evaluation, verifying with more detail how the experiments and experimentation in Chapter 7 relate and validate our research hypotheses in Section 8.5.

### 8.1 SPARQL<sub>Stream</sub> Evaluation

We have performed a series of evaluation experiments with SPARQL<sub>Stream</sub>, aimed at:

- Assessing the potential overhead of query rewriting during query execution, for both push and pull delivery scenarios (Hypothesis H5).



- Showing the feasibility of the query rewriting of ontology-based queries over DSMS, CEP or sensor middleware, using RDB2RDF mappings (Hypothesis H3).

### 8.1.1 Comparing the Query Execution w/wo Rewriting

We have provided implementations for four different systems with diverse characteristics and designed for different purposes. For instance, SNEE is capable of executing joins with stored data, but does not support push delivery. Cosm has a wide range of available data streams although is limited in query expressivity. GSN offers more query operators, but is also limited, for instance in the case of joins between streams. Esper offers event pattern matching, but does not allow union operators and joins in pull mode. It is not our goal to evaluate our approach with all these systems and to compare them exhaustively, given their large heterogeneity. Instead we will focus on the evaluation of the main characteristic of our system: the query rewriting and data translation steps that we add to the processing stack.

Therefore, we analyze the overhead added by these steps, so as to assess their potential impact in real-world scenarios. In order to do that we have evaluated both the pull and push based delivery mechanisms, with Esper as DSMS, since it is mature and stable, and provides both delivery modes. Since our framework delegates the query processing to the DSMS, we did not cover query complexity in the evaluation, hence we limited the tests to simple queries.

In many of these experiments, the absence of a established standard for streaming RDF and SPARQL querying makes it difficult to compare our approach with others or even with traditional DSMS or CEP. Even under these circumstances we performed our experiments using both CEP (Section 8.1.1) and DSMS (Section 8.1.2). Moreover, as we will see in Section 8.3 we even introduced the first such benchmark in the community, to allow more and better community-wide comparisons.

All these tests have been performed on an Intel Core i7 1.60 GHz, 6 GB.

**Rewriting and Translation Overhead in Pull Delivery.** In this experiment our objective is to assess the overhead caused by the query rewriting and data translation steps, during pull-based queries. We evaluated the response time to pull requests every 100 milliseconds, using a simple SPARQL<sub>Stream</sub> query equivalent to the one in Listing 8.1.

```
SELECT ?windspeed
FROM NAMED STREAM <http://swiss-experiment.ch/data#WannengratSensors.srdf>
[NOW-10 MINUTE TO NOW-0 MINUTE]
```

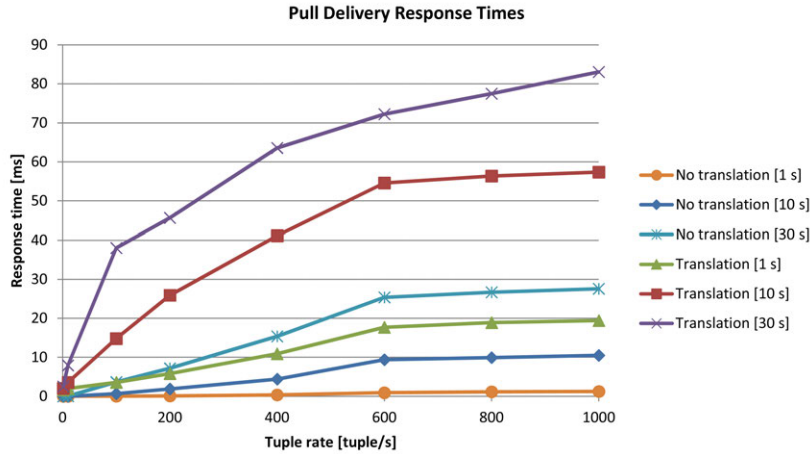
```

WHERE {
  ?WaveObs a ssn:Observation;
  ssn:observationResult ?windspeed;
  ssn:observedProperty sweetSpeed:WindSpeed.
}

```

**Listing 8.1:** SPARQL<sub>Stream</sub> query for all wave height sensors

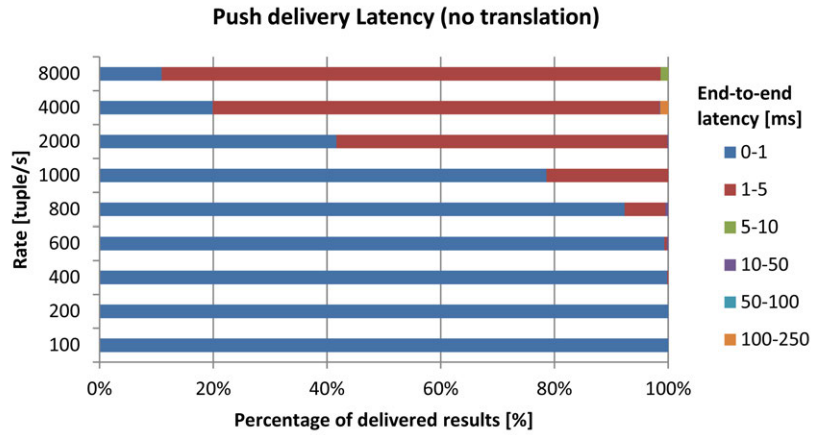
The system was loaded with 30 pre-configured synthetic streams, fed at the specified rate by a tuple generator (1 to 1000 tuples/s). We compared these results to pull requests to the equivalent EPL query, using Esper’s API directly, without query rewriting nor data translation steps. Since the query is translated only once, the real interest of this experiment is to verify the overhead of the data translation process. We experimented with three different time windows (1, 10 and 30 seconds), because the time boundaries significantly affect the number of tuples that are retrieved and translated to SPARQL results. As we can see in Figure 8.1, the executions with data translation have a significant overhead which is more noticeable as the tuple rate increases. As expected, the translation overhead depends directly of the number of tuples that the query is handling, and this may depend either on the tuple rate or the time window. Nevertheless, even for the relatively high rates we obtained acceptable response times.

**Figure 8.1:** Pull response times for different tuple rates and windows.

**Rewriting and Translation Overhead in Push Delivery.** Esper provides its own benchmark for performance evaluation in push delivery, which is free to be used and modified. We focused on the end-to-end latency of the generated tuples, which is featured in this benchmark, comparing the results of executing Esper EPL queries without

our framework and then with the query and data translation mechanism in place. For both cases we experimented with 100 to 8000 data values per second, and we plotted the results in Figure 8.2 and Figure 8.3 respectively, grouping the messages by latency ranges (as indicated in the Esper benchmark).

As expected, we see higher end-to-end latency for executions with our translation mechanism. For instance for 600 tuples/s, the original version has almost all messages under the line of 1 ms of latency. On the contrary, our implementation has most results between 1 and 5 ms. Considering that query translation is performed only once, most of this penalty comes from the data translation process, which could be optimized for push delivery, for instance by processing data in batches. Even with these limitations, for low and medium throughput requirements (e.g. 1-100 tuples/s), such as the case of the Swiss-Experiment environmental sensors, this component is comparable to the version without translation.

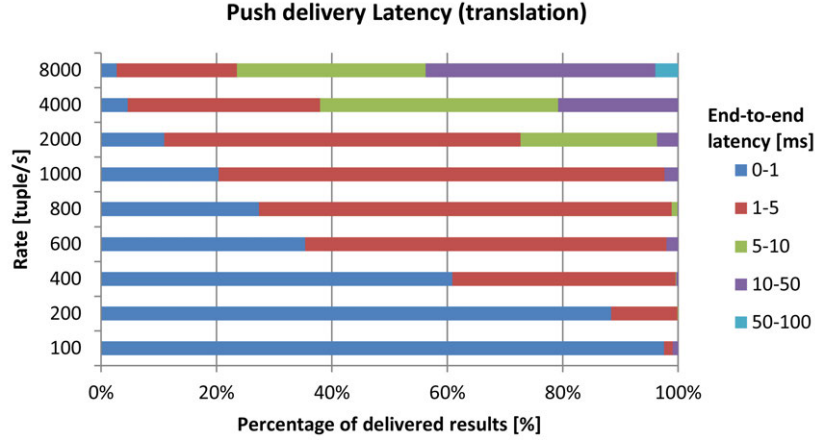


**Figure 8.2:** Push-delivery latency, without query/data translation.

### 8.1.2 Evaluation of SPARQL<sub>Stream</sub> Performance

In this section we evaluate the execution of SPARQL<sub>Stream</sub> queries when subject to medium-to-high tuple rates, window slides and increasing number of concurrent continuous queries. In particular, the evaluation of the experiments in this section were performed using the SNEE DSMS. This reinforces the evidence of applicability of SPARQL<sub>Stream</sub> for different underlying systems (H4).

The tests reported have been performed using synthetic data streams that use a tuple generator, configured with the desired settings. These synthetic streams emulate the sensor names and data types and can be easily tuned and parametrized.



**Figure 8.3:** Push-delivery latency, with query/data translation.

The parameters we used are:

- The number of deployed sensors: We used a fixed number of 25 wave sensors, mimicking those we have in the CCO SemSorGrid4Env (Gray et al., 2011a) deployment in the south coast of England. We also have 7 sensor streams for the tide sensors.
- The number of concurrent queries launched: In our system queries are typically continuously run, i.e. they get results periodically, so that a client can pull the latest result sets. The number of queries launched can affect the query response time as they are simultaneously fetching data from the streams.
- The stream rates: each sensor produces data tuples at some rate. If the rate is high, e.g. 1 tuple per second, the query response time can be affected, specially because of data translation.
- Time windows: time windows specify the time range of the stream tuples that will be included in the query.

A sample synthetic stream logical schema is given in Listing 8.2. Its corresponding physical schema is given in Listing 8.3. Both SNEE schemas are described in Calbimonte et al. (2011b).

```
<stream name="envdata_milford" type="push">
  <column name="DateTime"> <type class="timestamp"/></column>
  <column name="timestamp"> <type class="timestamp"/></column>
  <column name="stream_name"> <type class="string"/> </column>
  <column name="Hs"> <type class="float"/> </column>
```

```

    <column name="Hmax">          <type class ="float"/>  </column>
    <column name="Lat">           <type class ="float"/>  </column>
    <column name="Lon">           <type class ="float"/>  </column>
</stream>

```

**Listing 8.2:** Sample sensor logical schema in SNEE

```

<extent name="envdata_milford">
  <push_source>
    <port>6802</port>
    <rate>1000</rate>
  </push_source>
</extent>

```

**Listing 8.3:** Sample physical stream schema in SNEE: the rate denotes the tuples per second

All tests have been performed on an Intel Core i7 1.60 GHz, 6 GB.

**Evaluation of Execution over Medium-high Tuple Rates.** In this experiment we considered a simple query that requests the wave height values in one sensor location. The query is specified in SPARQL<sub>Stream</sub> below in Listing 8.4.

```

PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX ssn: <http://purl.oclc.org/NET/ssnx/ssn#>
PREFIX cd:  <http://www.semsorgrid4env.eu/ontologies/CoastalDefences.owl#>
PREFIX ssg: <http://semsorgrid4env.eu/ns#>

SELECT ?wavets ?waveheight
WHERE {
    ?WaveObs a ssn:Observation;
              ssn:observationResultTime ?wavets;
              ssn:observationResult ?waveheight;
              ssn:observedProperty cd:WaveHeight;
              ssn:observedBy ssg:MilfordSensor.
}

```

**Listing 8.4:** SPARQL<sub>Stream</sub> query requesting the wave heights of the Milford sensor

As it is specified by the last triple pattern, only those measurements observed by the ssg:MilfordSensor will be considered in the query. Following the mapping definition for the Milford sensor, the query is translated to the SNEEq<sub>l</sub> expression in Listing 8.5.

```
(SELECT Hs AS waveheight, DateTime AS wavets FROM envdata_milford) ;
```

**Listing 8.5:** Translated query in SNEEq<sub>l</sub>

We also analyzed the behavior when we change the SPARQL<sub>Stream</sub> query and eliminate the `ssg:MilfordSensor` restriction. Then we end up with the following SPARQL<sub>Stream</sub> query (Listing 8.6) and its corresponding SNEEqL translation (Listing 8.7).

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX ssn: <http://purl.oclc.org/NET/ssnx/ssn#>
PREFIX cd: <http://www.sensorgrid4env.eu/ontologies/CoastalDefences.owl#>
PREFIX ssg: <http://sensorgrid4env.eu/ns#>

SELECT ?wavets ?waveheight
WHERE {
    ?WaveObs a ssn:Observation;
        ssn:observationResultTime ?wavets;
        ssn:observationResult ?waveheight;
        ssn:observedProperty cd:WaveHeight;
        ssn:observedBy ?sensor.
}
```

**Listing 8.6:** SPARQL<sub>Stream</sub> query for all wave height sensors

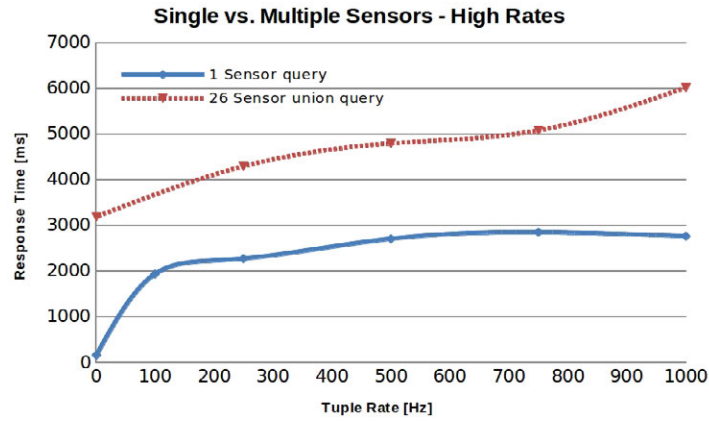
```
(SELECT Hs AS waveheight, DateTime AS wavets FROM envdata_bidefordbay) UNION
(SELECT Hs AS waveheight, DateTime AS wavets FROM envdata_boscombe) UNION
(SELECT Hs AS waveheight, DateTime AS wavets FROM envdata_bracklesham) UNION
(SELECT Hs AS waveheight, DateTime AS wavets FROM envdata_chesil) UNION
...
(SELECT Hs AS waveheight, DateTime AS wavets FROM envdata_wavenet_poolebay) ;
```

**Listing 8.7:** Translated SNEEqL query requesting all wave heights in a UNION. Shortened for readability

In this new scenario the query accesses 26 sensors simultaneously. We executed a continuous query under both settings and measured the response times, with high tuple update rates: up to 1000 tuples per second for each sensor. The results, comparing both the single sensor and 26-sensors are depicted in Figure 8.4.

As the query including the 26 sensors is much more complicated, the query is expected to take more time. However the main explanation for this delay is that it is a union query, hence it includes more data that will pass through the data translation process into ontology instances (SPARQL bound variables to be precise in this case).

**Evaluation of Queries with Different Window Slides.** Queries in SPARQL<sub>Stream</sub> allow specifying time windows that limit the tuples to be considered in the query, based on temporal constraints specified by the window boundaries. For instance the query in



**Figure 8.4:** Response times for a single-sensor query, compared to a 26 sensor union query, for different tuple rates

8.8 is a slight modification of query in Listing 8.4, with the addition of a 10 minute time window. The slide parameter indicates how often is the window computed (e.g. every 10 seconds). The query is translated as in Listing 8.9.

```
PREFIX sb: <http://www.w3.org/2009/SSN-XG/Ontologies/SensorBasis.owl#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX ssn: <http://purl.oclc.org/NET/ssnx/ssn#>
PREFIX cd: <http://www.sensorgrid4env.eu/ontologies/CoastalDefences.owl#>
PREFIX ssg: <http://sensorgrid4env.eu/ns#>

SELECT ?wavets ?waveheight
FROM NAMED STREAM <http://sensorgrid4env.eu/ns#ccometeo.srdf>
[NOW - 10 MINUTE SLIDE 10 S]
WHERE {
    ?WaveObs a ssn:Observation;
        ssn:observationResultTime ?wavets;
        ssn:observationResult ?waveheight;
        ssn:observedProperty cd:WaveHeight;
        ssn:observedBy ssg:MilfordSensor.
}
```

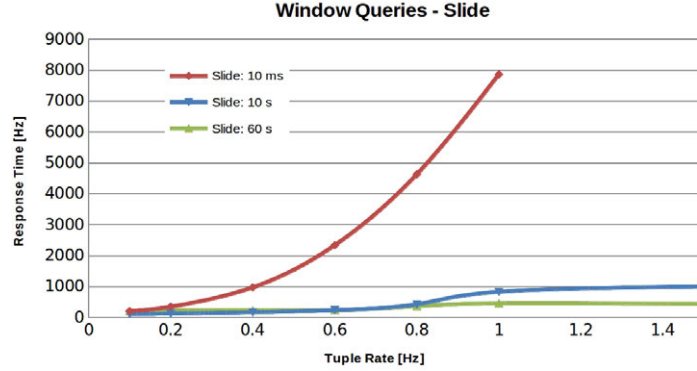
**Listing 8.8:** SPARQL<sub>Stream</sub> query with a time window

```
(SELECT Hs AS waveheight, DateTime AS wavets
FROM envdata_milford[FROM NOW - 10 MINUTES TO NOW - 0 MINUTES SLIDE 10 SECONDS] envdata_milford) ;
```

**Listing 8.9:** Translated SNEEqL with time window

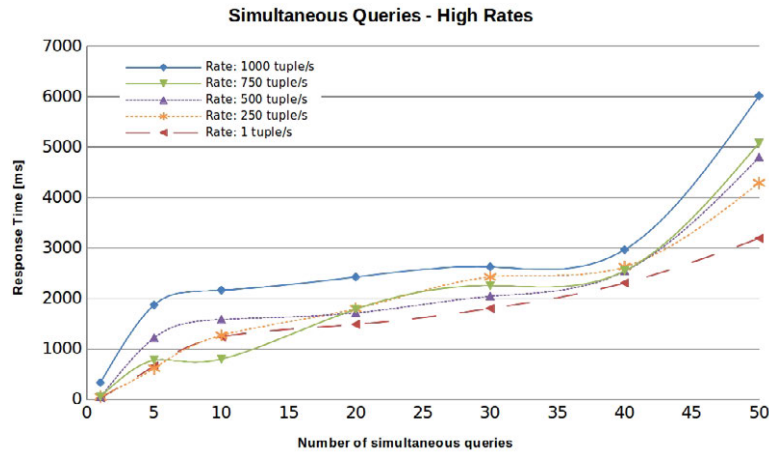
We can vary the window slide (i.e. how often the window is constructed). We argue that very frequent sliding may increase the response times significantly. The following

experiment was conducted to verify this. We executed queries with different slides: 60 seconds, 10 seconds and 10 milliseconds, as in Figure 8.5. As we can see, as the slide decreases, the query response time grows drastically. Even with relatively low rates, with a 10 ms slide the query quickly becomes unresponsive.



**Figure 8.5:** Response times for time window queries, with different slide parameters

**Evaluation for Concurrent Continuous Queries.** In our approach, the user typically issues a query once, and it gets executed continuously, so that results can be fetched upon a request. As the number of simultaneous queries increases, the query response time of each individual query may be affected, specially if the tuple rate is very high. In the following experiment we experimented with high tuple rates (1 to 1000 tuples per second), with 1 to 50 simultaneous queries. The results are depicted in Figure 8.6.



**Figure 8.6:** Response times for different number of queries launched simultaneously, for different tuple rates



With these very high rates, we can see that the service responds in around 2 seconds until we reach the number of 50 simultaneous queries. at this point the response time increases importantly (for all rates), and beyond that number of queries the service get stuck and out of memory. Notice that rates above 1 tuples per second already show signs of low response times altogether.

### 8.1.3 SPARQL<sub>Stream</sub> Evaluation Discussion

The main goal of this section was to assess the overhead of query rewriting and data translation in SPARQL<sub>Stream</sub> (hypothesis H5). We focus on this issue because SPARQL<sub>Stream</sub> offers semantic exposure of underlying data sources at the price of these additional processes in the query stack. We showed in Section 8.1.1 that we can achieve acceptable response times and latency in both pull and push delivery modes.

Moreover we reinforced the validity of hypothesis H3, about the feasibility of query rewriting using RDB2RDF mappings, by showing how it performs under different settings in terms of number of queries, window slides and rate variations of a sensor data deployment as the one in the SemSorGrid4Env project. Although H3 is already verified by the theoretical foundations described in Chapter 4, we also show evidence of its implementation feasibility.

## 8.2 Comparison of the SPARQL<sub>Stream</sub> Expressiveness with other Languages

The general query rewriting approach of SPARQL<sub>Stream</sub> uses streaming processing engines for query evaluation. These may use different query languages as seen in Chapter 2, with different expressive power. We evaluate SPARQL<sub>Stream</sub> in terms of expressiveness, compared to different representative DSMS, CEP and middleware API query languages. This evaluation supports evidence for validating hypothesis H2 and to a lesser extent H4, as it will be seen in Section 8.5.

### 8.2.1 DSMS: SNEEql and CQL

Borrowing heavily from DSMS query languages, SPARQL<sub>Stream</sub> is designed to cope with most of their streaming operators because it is rewritten into relational-based algebra expression with streaming constructs. CQL is a representative language of this kind, while SNEEql is a closely-related implementation that we used in the experimentation and evaluation.

We can enumerate the different query features as follows:

**Projection.** Projecting variables in CQL or SNEEql follows exactly the SQL semantics and is directly supported by standard SPARQL queries and hence by SPARQL<sub>Stream</sub>.

**Projection Expressions.** Apart from variables CQL and SNEEql support arithmetic and other operations in the selection expressions. These are equivalent to projection expressions in SELECT queries in SPARQL<sub>Stream</sub>, following SPARQL 1.1, e.g. SELECT (?value+5) AS ?newvalue.

**Join.** SPARQL joins (AND) between patterns may be translated to relational joins in relational streaming queries. However, in some cases triple pattern joins can be answered with a single stream, in which case the join simplification optimizations will eliminate the unnecessary join expressions. Joins may also include relational static data, which corresponds to joins between a stream graph and a static RDF graph in SPARQL<sub>Stream</sub>, i.e. FROM and FROM NAMED STREAM respectively. Nevertheless, languages as SNEEql are limited to joins only between windows, and not unbounded streams. Outer joins can be implemented with OPTIONAL patterns.

**Union.** Unions in relational stream queries are analogously supported with SPARQL UNION. However, SPARQL<sub>Stream</sub> non-union queries might be translated to relational stream queries with union operators if the mappings contain multiple definitions that fit a triple graph pattern. For instance given two R2RML subject mappings having a rr:class ssn:Sensor, then a triple pattern ?s rdf:type ssn:Sensor would match both of them. However, in SNEEql unions are not possible to apply between windows.

**Filter.** Filters in SPARQL<sub>Stream</sub> are in general translated as relational filters in the target language. However, there exist cases in which a pattern may be rewritten as a filter, e.g. in case of a constant object triple pattern.

**Aggregates.** Aggregate functions are covered by SPARQL<sub>Stream</sub> as it extends specification of SPARQL 1.1.

**Time Windows.** Time windows in SPARQL<sub>Stream</sub> can be rewritten to windows in CQL or SNEEql, including time ranges and slide parameters. To determine to which stream the window should be applied, the mappings R2RML rr:graph is used.

**Tuple Windows.** Tuple windows in CQL or SNEEqL do not have an equivalence in SPARQLStream. Although it would be tempting to include a triple-based window, it does not produce equivalent results, because the number of triples is not necessarily the same number of corresponding tuples in the source stream. Even if a ratio can be established (e.g. for 1 tuple there are 3 triples), the unordered nature of streaming tuples does not guarantee that the expected results are obtained. For this reason we omitted triple windows, while most of its use can be implemented with time windows and limits.

**Window-to-Stream operators.** SPARQLStream borrows the notion of ISTREAM, DSTREAM, and RSTREAM from the equivalent operators in CQL, also available in SNEEqL.

### 8.2.2 CEP: Esper

Esper includes the aforementioned features, and adds some others that are not available in DSMS and are common in CEP. Because SPARQLStream is based on DSMS operators, it does not include them in the language specification. These include output modifiers, repeating patterns, conjunction and disjunction of events, and sequences.

**Output modifiers.** Examples of modifiers are *first* and *last*, that output only the first matching or the last event matching a query, respectively.

**Repeating patterns.** An event *E* repeating a number of times *n* can be represented as *[n] E* in Esper. Although it can be a useful operator, it can be simulated to some extent using a counting aggregate function in SPARQLStream.

**Conjunction and disjunction.** The simultaneous presence of two events can be expressed as a conjunction in Esper: e.g. (*A* and *B*). In SPARQLStream a triple is not necessarily equal to an event, but in general this can be simulated using graph pattern joins. As for the Esper disjunction (e.g. (*A* or *B*)), the usage of optional joins can produce an equivalent query.

**Sequences.** A sequence is a pattern that represents an event followed by another one, e.g. (*A* -> *B*). There is no equivalent construct in SPARQLStream, although other languages, notably EP-SPARQL introduce similar operators.

### 8.2.3 Middleware APIs: GSN and Cosm

Although middleware are accessible through APIs and not proper query languages, we can find equivalences between them. As we will see, most of these APIs provide very limited expressiveness.

**Projection.** The GSN API allows choosing the attributes from the sensor streams that will be included in the query. SPARQL<sub>Stream</sub> projections are rewritten to this operation, although there are more complex scenarios that are not supported, e.g. expressions in the projections. In the Cosm API it is not possible to limit the attributes of a datastream. All attributes are output, therefore SPARQL<sub>Stream</sub> projections cannot be translated and are ignored when the Cosm request is generated. However this does not prevent from executing the query.

**Join.** Joins between streams are not supported on the GSN or Cosm REST APIs. However, in GSN is it possible to configure a virtual sensor that essentially computes a view over other virtual sensors, and in this case a join can be performed. Otherwise, a SPARQL<sub>Stream</sub> query that is translated into a join algebra operator, is not possible to be rewritten as a request in these middleware. This also applies for outer joins.

**Aggregates.** Although the GSN API is stated to feature aggregate functions, these are not fully operational on the available versions. In Cosm aggregates are not available at all. For this type of SPARQL<sub>Stream</sub> queries, the only alternative is to implement the aggregation as a post-processing operation after the data has been retrieved from the sensor middleware.

**Filters.** GSN allows a limited number of data filters, mainly targeting numeric comparisons. In Cosm no filters are available at the datastream level, but only for metadata. Again, in these cases SPARQL<sub>Stream</sub> queries are either not translatable or must implement filtering at post processing, which can be very inefficient.

### 8.2.4 Expressiveness Discussion

We have discussed the expresiveness of SPARQL<sub>Stream</sub> queries wrt. the underlying query languages, either fro DSMS, CEP or middleware. Although we did not elaborate an exhaustive comparison of languages available in the literature, we picked representative ones, with which we actually experimented, as detailed in Chapter 7. We identified some

of the salient features and operators in these languages, and showed evidence that most of them are covered by SPARQL<sub>Stream</sub>, therefore allowing the query translation mechanism. This enables the use of such language extensions for querying streaming data, as seen in hypothesis H2.

Moreover, we evidenced that in some cases, notably for middleware APIs, not all SPARQL<sub>Stream</sub> queries are directly rewritable, because the former lack a number of operators on its specification. Nonetheless, SPARQL<sub>Stream</sub> could be extended for covering some of the operators available exclusively in CEP such as Esper, but this work is outside the scope of this thesis. therefore the expressiveness coverage of SPARQL<sub>Stream</sub> supports the validity of hypothesis H4, as it covers language features from these different type of systems. Finally, we present a summary of this comparison in the following Table 8.1.

Feature	SPARQL <sub>Stream</sub>	CQL	SNEEql	Esper EQL	GSN API	Cosm API
Projection	Yes	Yes	Yes	Yes	Yes	Fixed
Projection expression	Tuple Window	Yes	Yes	Yes	No	No
Joins	Yes	Yes	Only windows, no outer	Yes	No	No
Union	Yes	Yes	Not for windows		Yes	No
Aggregation	Yes	Yes	Yes	Yes	Limited	No
Selection	Yes	Yes	Yes	Yes	Yes	No
Time Window	Yes	Yes	Yes	Yes	Yes	Limited
Tuple Window	No	Yes	Yes	Yes	Yes	No
Window-to-Stream	Yes	Yes	Yes	Yes	No	No
Output modifier	No	No	No	Yes	No	No
Conjunction, disjunction	With Joins	No	No	Yes	No	No
Repetition pattern	With Count	No	No	Yes	No	No
Sequence	No	No	No	Yes	No	No

**Table 8.1:** Comparison of SPARQL<sub>Stream</sub> expressiveness wrt. CQL, SNEEql, Esper, GSN and Cosm.

### 8.3 Functional Evaluation of SPARQL<sub>Stream</sub>

As we have seen in Section 2.3, there is an increasing interest in streaming RDF/SPARQL engines, although each has its own set of extensions and very different implementation approaches. Therefore we were presented with the need for a standard way to compare the functionality and performance such systems. Little work on benchmarking streaming data engines has been produced in the literature. In fact, for DSMSs only the Linear Road benchmark (Arasu et al., 2004b) is available, although it is not adequate to assess RDF streaming engines. Instead, it is designed to evaluate traditional DSMS based on the relational data model. Moreover, Linear Road overlooks important aspects for related to RDF data such as interlinking and reasoning. Existing SPARQL benchmarks

such as (Bizer and Schultz, 2009; Guo et al., 2005; Schmidt et al., 2009), are focused exclusively on static data, so they do not capture the dynamic properties of streaming data.

In consequence, we designed SRBench (Zhang et al., 2012a), a streaming RDF/SPARQL benchmark that aims at assessing the abilities of streaming RDF engines in dealing with a broad range of different query types in which Semantic Web technologies, including querying, interlinking, sharing and reasoning, are applied on highly dynamic streaming RDF data. SRBench is the first general-purpose benchmark that is primarily designed to compare streaming RDF/SPARQL engines.

The design of SRBench was based on the principle of addressing the following challenges: the need for a proper streaming RDF dataset, the definition of a concise set of features and the absence of a single SPARQL based streaming query language.

**Benchmark Data set.** The design of a streaming RDF/SPARQL benchmark requires a cautiously chosen data set that is relevant, realistic, semantically valid and interlinkable (Zhang et al., 2012a). Additionally, the data set should allow the formulation of queries that both feel natural and present a concise but complete set of challenges that streaming RDF engines should meet. Considering the study of Duan et al. (2011), which points out that the synthetic data does not predict how RDF stores behave in realistic scenarios, and the inherent importance of interlinking RDF data, we used real-world data sets for SRBench. Specifically, we used LinkedSensorData (Patni et al., 2010), the largest sensor dataset from the Linked Open Data cloud<sup>1</sup>, which includes both observations and sensor metadata. To assess a system’s ability of dealing with interlinked data, we additionally use the LOD data sets GeoNames<sup>2</sup> and DBpedia<sup>3</sup>, which are linked to the LinkedSensorData.

**A concise set of features.** Applying Semantic Web technologies to streaming data helps providing explicit semantics to data for search and reuse, may enable reasoning through ontologies, and facilitates the integration with other data sets. The benchmark provides a comprehensive set of queries that assess a system’s ability of processing these distinctive features on highly dynamic (in terms of arriving rate and amount) streaming data, possibly in combination with static data. Following the “20 queries” principles (Hey et al., 2009), in SRBench we defined seventeen queries that have been carefully cho-

---

<sup>1</sup>Linked Data: <http://linkeddata.org/>

<sup>2</sup>GeoNames Ontology: <http://www.geonames.org/ontology/>.

<sup>3</sup>DBpedia datasets: <http://wiki.dbpedia.org/Datasets>

sen such that they provide valuable insights that can be generally applied to streaming RDF systems and are useful in many domains, e.g., notion of time bounded queries (e.g., data in the latest X time-units); notion of continuous queries (i.e., queries evaluated periodically); data summarization in the queries (e.g., aggregates); providing high-level abstractions from raw-data ; and combining streams with contextual static data.

**No standard query language.** No standard has been established yet for streaming data processing, and even less for streaming SPARQL extensions. Therefore, the queries of a streaming benchmark should be specified in a language agnostic way, yet have a clear semantics. In SRBench, this challenge is met by providing a descriptive definition of each of the benchmark queries. Then, we provide implementations of the benchmark queries using the three major SPARQL extensions for streaming data processing, i.e., C-SPARQL, CQELS and our own SPARQL<sub>Stream</sub>. These three sets of implementing queries are used for the purpose of clarifying the benchmark query definitions.

### 8.3.1 SRBench Queries

The 17 queries<sup>1</sup> of SRBench cover a wide range of features from RDF and SPARQL processing, more advanced SPARQL 1.1 features, inherent streaming data features available in DSMS, and interlinking with external static data. The features considered are discussed below.

**Graph pattern matching.** An essential feature of SPARQL queries, it includes features such as the basic graph pattern matching operators `'` (representing a natural join AND) and `FILTER`, and the most complicated operators `UNION` and `OPTIONAL` (Pérez et al., 2009). SRBench queries includes all these operators.

**Solution modifiers.** In SRBench, only the projection and `DISTINCT` solution modifiers are addressed, because the additional values of the other four operators are negligible in streaming applications. `ORDER BY` is ignored since streaming data are already sorted by their time stamps, and sorting the results on another attribute will only produce partially sorted data (within one window). The features of `OFFSET` and `LIMIT` are largely covered by sliding windows, which are more appropriate for streaming queries. Finally, the nondeterministic nature of `REDUCED` highly complicates the verification of the query results.

---

<sup>1</sup>Full descriptions and implementations of the queries are available at <http://www.w3.org/wiki/SRBench>

**Query forms.** SRBench supports the 3 main query forms of SPARQL: SELECT, CONSTRUCT and ASK. The other query form DESCRIBE returns an RDF graph that describes the resources found. In the SRBench queries, this form is not used, because it is highly implementation dependant, which largely complicates the verification of the query results.

**SPARQL 1.1.** The additions to the SPARQL language included in the SPARQL 1.1 W3C Proposed Recommendation introduced several new features, including aggregates, sub-queries, negation, projection expressions, Property Paths and assignment. We make extensive use of these new features in SRBench, especially aggregates and property paths, although the semantics of the latter might be modified, due to recent analysis of the unfeasibility of their current semantics ([Arenas et al., 2012](#)).

**Reasoning.** SRBench includes queries that allow exploiting reasoning if provided by the processing engine. Currently, the queries involve reasoning over the `rdfs:subClassOf`, `rdfs:subPropertyOf` and `owl:sameAs` properties. The queries in SRBench can be implemented and executed by both systems with and without inference mechanisms, but the differences might be noticeable in the query results. Also note that, although SPARQL is not a reasoning language, it can be used to query ontologies if they are encoded in RDF. So, on systems without reasoning, this shortcoming can be alleviated by explicitly expressing some reasoning tasks using extra graph patterns with Property Path over the ontologies.

**Streaming.** Existing streaming SPARQL extensions generally introduce streaming data operators inspired by DSMS continuous query languages such as CQL (see Section 2.1.2). We considered three important streaming SPARQL features: the time-based sliding window and the window-to-stream operators. Also, the continuous nature of queries is considered in SRBench, as an orthogonal feature to all operators mentioned so far.

In Table 8.2 we provide a summary of these features in the 17 SRBench queries.

#### 8.3.2 Functional Evaluation

Streaming RDF/SPARQL query engines are relatively new and are mostly in the beginning stage of development. Therefore we deemed it appropriate to conduct a functional evaluation of these systems, aiming at finding out if the functionalities they provide are sufficient for realistic streaming query processing, if there are any crucial features missing or if there are distinctive operators that difference one system wrt. the others. The proposed evaluation includes three of such systems: SPARQL<sub>Stream</sub>, C-SPARQL and



	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	Q11	Q12	Q13	Q14	Q15	Q16	Q17
1. Graph pattern matching	A	A,F,O	A	A,F	A	A,F,U	A	A	A	A	A,F	A,F,U	A,F	A,F,U	A,F	A,F	A,F
2. Solution modifier	P,D	P,D	P	P	P	P	P,D	P	P	P,D	P,D	P	P	P,D	P	P	P
3. Query form	S	S	A	S	C	S	S	S	S	S	S	S	S	S	S	S	S
4. SPARQL 1.1		F,P	A	A,E, M,F	A,S		N	A,E,M	A,E,M		A,S, M,F	A,S,E, M,F,P	A,E,M, F,P	F,P	A,E,M,P	P	P
5. Reasoning		C	R												C	A	C
6. Streaming	T	T	T	T	T	T	T,D	T	T	T	T	T	T	T	T		
7. Dataset	O	O	O	O	O	O	O	O,S	O,S	O,S	O,S	O,S,G	O,S,G	O,S,G	O,S,D	O,S,G,D	S

**Table 8.2:** Addressed features per query. Operators abbreviations: 1. **A**nd, **F**ilter, **U**nion, **O**ptional; 2. **P**rojection, **D**istinct; 3. **S**elect, **C**onstruct, **A**sk; 4. **A**ggregate, **S**ubquery, **N**egation, **E**xpressions, assign**M**ent, **F**unctions, **P**roperty path; 5. sub**C**lassOf, sub**P**ropertyOf, owl:sameAs; 6. **T**ime-based window, **I**stream, **D**stream, **R**stream; 7. Linked**O**bservationData, Linked**S**ensorMetadata, **G**eoNames, **D**bpedia.

CQELS, three representative approaches that provide available implementations. Although we do not mean to exhaustive, we consider that these systems already cover an interesting spectrum: SPARQL<sub>Stream</sub> relies on mappings and query rewriting over an underlying streaming engine, C-SPARQL divides execution between a SPARQL endpoint and a DSMS, and CQELS implements RDF stream processing from scratch. The evaluation results are intended to give a first baseline and illustrate the state-of-the-art, and with respect to this thesis, reinforce the validity of hypothesis H2, as we will see in Section 8.5.

To execute the queries, we downloaded the latest SPARQL<sub>Stream</sub><sup>1</sup>, CQELS (Aug. 2011) and C-SPARQL 0.7.4. All implementing queries can be found in the SRBench wiki page<sup>2</sup>. An overview of the evaluation results is shown in Table 8.3. A tick indicates that the engine is able to process a particular query. For each query that cannot be processed by a certain engine, we denote the main missing feature(s) that cause the query to fail.

System	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	Q11	Q12	Q13	Q14	Q15	Q16	Q17
SPARQLStream	OK	PP	A	G	G	OK	OK	G	G,IF	SD	SD	PP,SD	PP,SD	PP,SD	PP,SD	PP,SD	PP,SD
CQELS	OK	PP	A	OK	OK	OK	D/N	OK	IF	OK	OK	PP	PP	PP	PP	PP	PP
C-SPARQL	OK	PP	A	OK	OK	OK	D	OK	IF	OK	OK	PP	PP	PP	PP	PP	PP

**Table 8.3:** Functional evaluation results. Abbreviations of missing features: **A**sk, **D**stream, **G**roup and aggregates, **IF** expression, **N**egation, **P**roperty path, **S**tatic **D**ataset.

The evaluation results show that most of the basic SPARQL features are covered by all systems, i.e. graph pattern matching, SELECT and CONSTRUCT modifiers. Interestingly,

<sup>1</sup><https://github.com/jpcik/morph-streams>

<sup>2</sup><http://www.w3.org/wiki/SRBench>

a simple solution modifier such as ASK is not available in any of the chosen implementations. This detail evidences the early stage of development in which these engines are still immersed. The number of functionalities currently supported by SPARQL<sub>Stream</sub> is somewhat less than those supported by CQELS and C-SPARQL, mainly due to lack of support for GROUP BY operators.

Most of the other issues found in this evaluation are related to lack of support of SPARQL 1.1 features. Notably, none of the implementations support property paths, which prevents executing 7 out of 17 queries in SRBench. We regard the Property Path expressions as one of the most important features of SPARQL 1.1, because it provides flexible ways to navigate through RDF graphs and facilitates reasoning over various graph patterns. These are distinguishing factors of the usefulness of one system from others for streaming RDF applications.

Finally, in terms of reasoning, very little support is provided in current implementations. C-SPARQL is the only testing system that supports reasoning based on simple RDF entailment, although it is not available in the published implementation. SPARQL<sub>Stream</sub> and CQELS currently do not tackle the problem of reasoning. The overall conclusion of our evaluation is that there is no single best system yet, and although the SPARQL<sub>Stream</sub> engine supports fewer queries than CQELS and C-SPARQL, all of them still require covering features of SPARQL 1.1 and reasoning, and they also need reaching a more mature status in order to be used in real-world scenarios. Overall, we showed that SPARQL<sub>Stream</sub> and the other approaches (not based in query rewriting) are able to be used to query streaming data (hypothesis H2), but in terms of implementation there is still a gap to be filled. Nevertheless, this initial step towards a streaming RDF and SPARQL benchmark is a necessary contribution to allow better and comparable experimentation among these engines, although each one is focused on slightly different use cases.

## 8.4 Evaluation of the Sensor Data Characterization

The main goal of these experiments is to show that the proposed sensor data representation using slopes can be used to characterize sensor data and extract sensor metadata corresponding to the types of observed properties. First we show how the classification behaves with two real life data sets, in terms of precision. Next, we are interested in experimenting with smaller subsets of data samples, and observing how the classification behaves with less data, as we know there are repeating data patterns. Finally, we compare our approach with a classification using the widely used SAX symbolic representation of the data (Lin et al., 2007).

To validate the classification approach presented in Section 6.3.2, we implemented and applied it to two different datasets in the environmental domain: one from the Swiss Experiment<sup>1</sup> and another from AEMET. The data is heterogeneous as it comes from different geographical locations, some have different time spans (e.g. observations collected during 1 year, 3 months, etc), others have different sampling rates. Also the number of sensors per observation type varies (e.g. 78 for temperature, only 4 for snow height). Due to the conditions of the deployments, some of them experimental and others deployed in harsh environments, this dataset contains a considerable amount of noise in the data.

The AEMET dataset consists of sensor data from 100 weather stations managed by the Spanish meteorological office. The data is heterogeneous, coming from stations all over Spain, and was originally collected in intervals of 10 minutes. It contains, in general, less noise and anomalies than the Swiss Experiment dataset, as it comes from stations daily used for meteorological forecasts.

#### 8.4.1 Classification in Swiss Experiment and AEMET

The goal of our first experiment consists in evaluating the effectiveness of the classification in terms of precision and recall. The classifier is expected to assign the correct label (the type of observed property, e.g. “humidity”) to time series from a test set. The classifier uses a training set of time series and the evaluation criteria is computed in terms on the number of true positives ( $tp$ ), false positives ( $fp$ ) and false negatives ( $fn$ ): precision ( $p = \frac{tp}{tp+fp}$ ), and recall ( $r = \frac{tp}{tp+fn}$ ).

**Swiss Experiment.** The heterogeneity of the Swiss Experiment dataset required applying different parameters for the linear approximation step. Some time series had very short sampling time intervals (e.g. every 2 seconds for pressure, for at most two days), while others had very long ones (e.g. every half-an-hour for several months). Hence, the approximations were very different in these cases (hundreds of segments per day for short intervals, and only a few per day for long ones). We applied a 5-fold cross validation scheme to divide our dataset in training and test set, and then apply the nearest neighbor algorithm. We present the confusion matrix in Table 8.7, for  $k = 5$ .

We can observe that the effectiveness of the classification varies among the different types of data. The nearest neighbor scheme is also biased as the dataset is highly unbalanced. Since we have comparatively much more samples of temperature or wind speed,

<sup>1</sup>The dataset is available at: <http://lsirpeople.epfl.ch/qvnhguye/benchmark/>

#### 8.4. Evaluation of the Sensor Data Characterization

Match results Swissex k=5, 5-fold																	
test set	ra	mo	te	wd	ws	hu	ly	pr	co	sh	vo	total	fp	tp	fn	p	r
radiation	15		4	7	7						1	34	19	15	0	0.441	1
moisture		10	3	2	1						1	20	8	10	2	0.556	0.833
temperature	2	3	56		1	11					2	78	19	56	3	0.747	0.949
wind direction	4		1	25	4							35	9	25	1	0.735	0.962
wind speed			1	4	40	1						46	6	40	0	0.87	1
humidity	1		9		2	21						34	12	21	1	0.636	0.955
lysimeter		2					4					6	2	4	0	0.667	1
pressure								4				4	0	4	0	1	1
co2									10			11	0	10	1	1	0.909
snow height		1	2							1		4	3	1	0	0.25	1
voltage			6		1						9	16	7	9	0	0.563	1
<b>total</b>												<b>288</b>	<b>85</b>	<b>195</b>	<b>8</b>	<b>0.7</b>	<b>0.96</b>

**Figure 8.7:** Swiss Experiment confusion matrix, k=5. Column header abbreviations: ra: radiation, mo: moisture, te: temperature, wd: wind direction, ws: wind speed, hu: humidity, ly: lysimeter, pr: pressure, co: CO<sub>2</sub>, sh: snow height, vo: voltage

than for pressure or snow height, these last are less likely find nearest neighbors of the same class. For instance for *lysimeter* and *snow height*, almost no series are correctly identified, as we have a very small number of series. Nevertheless, in the cases of *pressure* or *CO<sub>2</sub>* the precision is good regardless of the low number of series. This is a special case, since these series have very different slope distributions, and also, have very short sampling interval. Since their resolution is much smaller (e.g. every 2 seconds) than most of the other series in the dataset, their comparison throws very large distances that are quickly discarded.

In cases where the total number of time series was very small (e.g. only 4 for *snow height*), the approach is clearly not effective. It requires a larger training set to have an acceptable precision. Also, when the series are very irregular (sometimes due to noise and false non-curved data in the original dataset), they logically fail to be correctly classified.

**AEMET.** For the AEMET dataset, we followed the same approach as with the Swiss-Experiment. However, for the AEMET data, we had a larger number of time series for every type of data, thus avoiding the problem of lack of training data encountered in the previous tests. Moreover, the dataset sampling interval is the same, making it easier to compare their slope distributions. We applied the classification scheme with a 10-fold cross validation for this dataset. We provide the confusion matrix for  $k = 5$  in Table 8.8.

We can notice that in this case the approach achieves better precision, as expected, since we avoided the problems of sampling times and unbalanced types (the number of series per each type is similar or the same). However, it can be observed that there are important false positives at some specific spots. For instance the number of *soil temperature* series falsely identified as *air temperature* is very high. This is in fact an expected result, since both are specializations of the more general type *temperature*. Hence, both

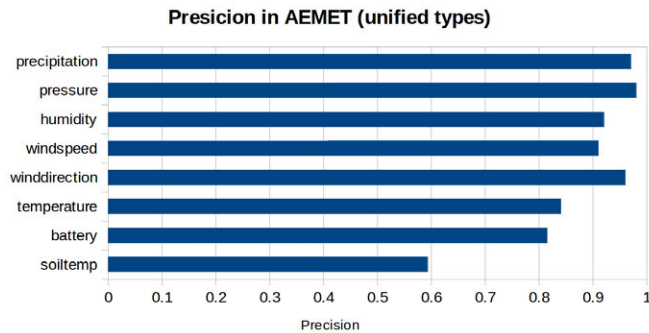
Match results AEMET k=5, 10-fold

test set	st	ba	te	wd	ws	hu	wsx	pr	wdx	pre	total	fp	tp	fn	p	r
soiltemp	48		23	1	1	2		6			81	33	48	0	0.59	1
battery	2	66		2		1				10	81	15	66	0	0.81	1
temperature	15	1	84								100	16	84	0	0.84	1
winddirection		1		43	3				53		100	57	43	0	0.43	1
windspeed		1	1	2	54	4	37		1		100	46	54	0	0.54	1
humidity	1		1		2	92	2		2		100	8	92	0	0.92	1
windspeedmax		1	1	1	54	3	39		1		100	61	39	0	0.39	1
pressure			2					97			99	2	97	0	0.98	1
winddirmax		1		43	3				53		100	47	53	0	0.53	1
precipitation		2					1			97	100	3	97	0	0.97	1
<b>total</b>											<b>961</b>	<b>288</b>	<b>673</b>	<b>0</b>	<b>0.7</b>	<b>1</b>

**Figure 8.8:** AEMET confusion matrix, k=5. Column header abbreviations: st:soil temperature, ba:battery, te:air temperature, wd:wind direction, ws:wind speed, hu:humidity, wsx: wind speed (max), pr:pressure, wdx: wind direction (max), pre:precipitation.

share patterns in the time series, that are reflected in the slope distributions that are compared during the classification process. The same situation can be seen between *wind speed* and *wind speed (max)*, and for *wind direction* and *wind direction (max)*.

It is also interesting to see that if we consider the “unification” of similar types of data (e.g. *wind speed* and *maximum wind speed*), the precision is much higher (Figure 8.9). This suggests that the slope distributions are useful for identifying similar data, because they have very similar slope distributions. This is an expected behavior, for instance for *wind speed* and *wind speed (max)*, which are measurements of the same type of data. In order to discern between small differences like these, other characteristics of

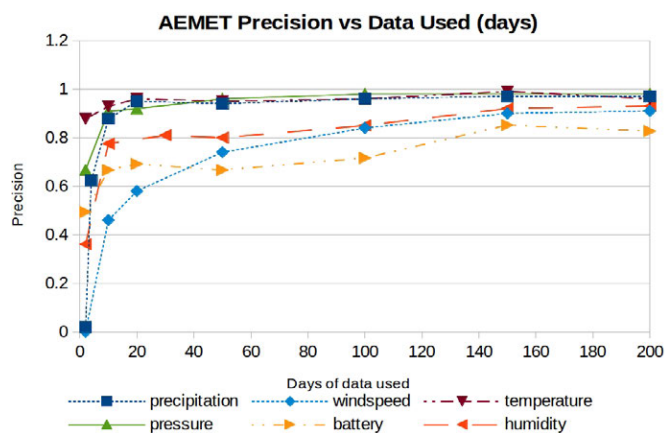


**Figure 8.9:** Precision in AEMET, not differencing the specific types wind speed (max) and wind direction (max).

the data have to be taken into account. In these cases where two types of observations are similar, we can use a higher level definition of observed property. For instance, in the Climate and Forecast vocabulary, the specific properties `cf-property:air_temperature` and `cf-property:soil_temperature` both have `qu:temperature` as its general quantity kind.

### 8.4.2 Classification with Partial Information

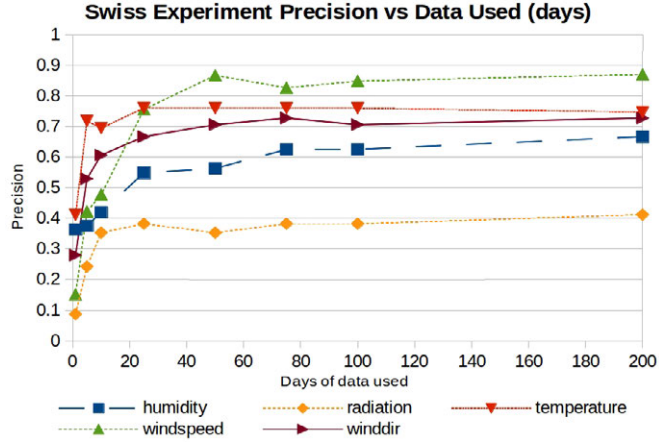
In this experiment we aim at showing how the classification precision varies when using smaller subsets of the test data. As we discussed in Section 6.3.3, for our environmental and meteorological datasets, recurrent slope patterns in the data can be representative enough to compute the slope distribution, and make it possible to classify the data. We have tested the classification reducing the number of days-of-data used for computation. In Figure 8.10 and Figure 8.11 we plot the precision for the AEMET and Swiss Experiment dataset series, for different subsets of the data (expressed in terms of the number of days of measured data). In total we have around 200 days of observations, but we can see that for some types of data we require much less and obtain similar precision in the classification. This is the case especially with series that include very repetitive patterns on a daily basis, but not for others that have a more unpredictable behavior such as *wind speed*. In this case we see that it needs more days-of-data than other types to increase the precision.



**Figure 8.10:** AEMET Classification precision, for different partial datasets, in terms of the days of data used.

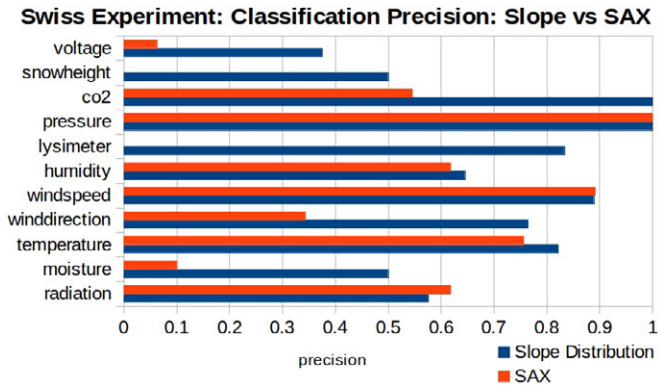
### 8.4.3 Comparison with SAX Classification

The goal of this experiment is to compare our approach with a classification based on the widely used SAX representation of time series (Lin et al., 2007). The comparison is based on the precision using both approaches. By classifying with SAX we can verify how well our method behaves in comparison to a well established technique. The SAX approach also produces a symbolization of the time series, although the angles and slopes are not taken into account, as it uses a PAA approximation. We applied the same classification



**Figure 8.11:** Swiss Experiment Classification precision, for different partial datasets, in terms of the days of data used.

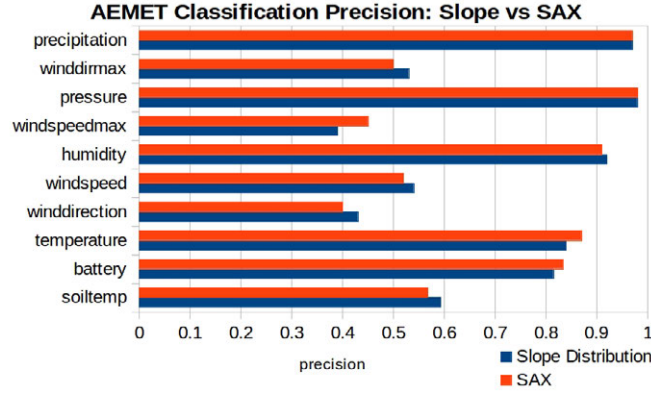
method used for our slope-based representation. We show the classification precision for the Swiss Experiment and AEMET datasets in Figure 8.12 and Figure 8.13 respectively.



**Figure 8.12:** Swiss Experiment Classification precision with SAX and the Slope representation.

As it can be seen, the classification throws similar results for both methods, with small differences in AEMET, and slightly better for the slope-based approach in the Swiss experiment dataset. Using the slopes distributions shows to be helpful at differentiating time series with similar values but very different angles. In the case of AEMET, the measured values are already enough to discern between two different types of observation, and hence the results are not improved by the slope distribution. While the SAX representation has been exploited in other ways, for example by considering substrings of a fixed size, instead of only one symbol, this experiment shows that our approach is also able to extract features that help characterizing a type of time series, and enabling





**Figure 8.13:** AEMET Classification precision with SAX and the Slope representation.

its semantic identification. A classification technique throws different results depending on the type of data. Further amendments could be plugged to the classification scheme, but they risk to be too specific to the characteristics of certain datatypes, and such methods are outside of the scope of this work.

#### 8.4.4 Discussion of the Characterization Evaluation

We evaluated the classification approach based on slope distributions with real-life data sets of the Swiss-Experiment project and AEMET. We have shown through experimentation that this representation can be useful for balanced datasets, as the classification gets biased when there are small numbers of samples in the training set, for a particular type of data. Moreover, our results show that this representation can help grouping data of the same type, despite geographical locations, since it is based on the distribution of slopes of a linear approximation. Therefore, it can identify similarities of related types of data: e.g. *air temperature* and *soil temperature*, validating hypothesis H6. We have compared our characterization of sensor data with a competitive approach, validating hypothesis H7, and showed that for the chosen environmental datasets it effectively enables the extraction of semantic metadata.

The proposed approach, however, was evaluated within the same dataset, and in the future we will study its applicability in an inter-dataset classification. This framework could be used in the future for other tasks such as clustering, or for identifying simple patterns in streams of sensor data. Moreover, complex symbolizations consisting of sequences of slopes could be considered, which would represent more complete patterns that can be exploited. Also, we can consider building a more complex representation that includes not only the slopes information but also the value ranges, and even tags



and labels provided the data publishers. This may enable a more complete and accurate extraction of metadata that enriches the growing Semantic Sensor Web. As a final future path, we may consider applying online execution of these techniques for real-time analysis.

## 8.5 Evaluation Conclusions

The work described in this thesis builds on the hypothesis presented in Chapter 3, addressing the corresponding research problems: P1: How to access streaming sensor data with continuous queries using ontology models to represent the sensor observations. P2: How to characterize sensor data using the recorded time series, capturing the semantic metadata of the sensor observations. Therefore, we proceed to validate each hypothesis with the evaluation results presented throughout this chapter and also the experimentation presented in the previous chapters.

**H1.** *Sensor streaming data can be treated as instances of concepts from a comprehensive ontology model, such as a combination of the SSN ontology with domain-specific ontologies.*

In Chapter 5 we presented the principles of mapping-based rewriting that allows querying sensor streaming data in terms of ontological concepts. In particular, we showed the feasibility of this approach in different use cases in Chapter 7, using the SSN ontology as a base model in conjunction with domain ontologies. In all these cases users and applications were able to treat sensor data as instances of ontology concepts using SPARQL-based queries.

**H2.** *SPARQL queries can be extended with streaming operators, and used to continuously query streaming data, in terms of the ontology concepts and properties.*

In Chapter 4 we presented SPARQL<sub>Stream</sub>, a formalization of streaming data operators for SPARQL, designed for ontology-based continuous query processing of streaming data. Along with the formal definition of the data model and language, we provided the semantics of SPARQL<sub>Stream</sub> and showed concrete examples of applicability and use in Chapter 7. We also compared the expressiveness of SPARQL<sub>Stream</sub> with different DSMS, CEP and middleware languages (Section 8.2), showing that it covers most features of these systems. We also provided a functional evaluation of SPARQL<sub>Stream</sub> compared with

other engines in Section 8.3, and established a technology benchmark for comparing this new type of query languages and their engines.

**H3.** *Ontology-based queries to streaming data can be rewritten into queries in terms of relational and event streams, and the query results can be returned as instances in terms of the ontologies. The use of existing relational-to-RDF (RDB2RDF) mappings can be extended to streaming data for this purpose.*

The semantics of query rewriting from SPARQL<sub>Stream</sub> to a relational stream algebra has been fully described in Chapter 4. We showed that existing relational and event streams can be accessed using the query rewriting mechanism, given a set of RDB2RDF mappings. Furthermore, we provided formal definitions of these mappings and how they are used during the rewriting process. Specifically, we showed in Chapter 5 that we can use the W3C Recommendation R2RML mappings for this purpose.

**H4.** *Ontology-based queries for streams can be expressed as abstract expressions that can be transformed into queries for relational streams and complex event processors, or sensor middleware service requests.*

We have extensively shown in Chapter 5 how the rewritten streaming algebra expressions can be instantiated as queries in different concrete query languages. These languages can be executed either by DSMS, CEP or sensor middleware. We have presented implementations and real use-case applications that use SPARQL<sub>Stream</sub> with query rewriting for these different technologies in Chapter 7, and the evaluation as well. We included the SNEE DSMS, the Esper CEP, and GSN and Cosm middleware as concrete implementation examples of our approach. We have also evidenced that some of these underlying languages and systems sometimes offer less expressiveness than SPARQL<sub>Stream</sub> in Section 8.2. Finally we provided empirical evidence of acceptable performance of continuous SPARQL<sub>Stream</sub> queries in medium-low rates in environmental sensor settings in Section 8.1.2.

**H5.** *Ontology-based query rewriting for streams can be applied for both push and pull data delivery, incurring in an acceptable overhead in query latency compared to native execution.*

We provided an evaluation of the query rewriting overhead of SPARQL<sub>Stream</sub> compared with native query execution in Section 8.1.1. We show that the overhead of query rewriting is acceptable even for medium tuple rates, both for push and pull delivery. We observed that even in the presence of overhead in query rewriting, the fact that continuous queries can be compiled only once, may greatly alleviate these penalties.

**H6.** *Sensor data series can be analyzed in order to find patterns in it, that characterize its type and makes it recognizable among other types of series.*

We have presented an approach in Chapter 6 that relies on sensor observations analysis to determine the type of observed property, given a training set of already classified series. We have experimented with two real-world data sets: AEMET and Swiss Experiment, and presented competitive and sometimes better results than with the alternative SAX representation in Section 8.4.3.

**H7.** *Sensor data time series representations can be analyzed, so that semantic properties such as the type of data, can be learned using mining clustering and classification techniques, with acceptable precision.*

We have shown that the classification method presented in Chapter 6 provides good precision in real-world datasets, especially for more homogeneous sensor time series such as the ones in AEMET (Section 8.4). We showed that the existence of already classified series influences the precision of our approach, and that for some types of data, even with only a small subset of the data we can already achieve high precision (Section 8.4.2).

## Chapter 9

# Conclusions

This thesis addresses different challenges in the area of streaming data access through semantic technologies. These challenges converge to the two main research problems we addressed in Chapter 3: (i) How to enable ontology-based access and querying over existing streaming data sources; and (ii) How to represent, classify and enrich semantic sensor metadata.

Although other approaches have been proposed, even concurrently to ours, for querying data streams with SPARQL-based query engines, they are either limited to a particular fixed model or technology, or implement query evaluation from scratch. Therefore none of the existing systems flexibly reuses streaming and event query processing engines, exposing the data through an ontological query interface. Moreover, because these systems may have very different data models and schemas, they need mappings that relate them to ontology concepts. None of the existing approaches considers using declarative mappings for these tasks, and in the area of RDB2RDF, streaming data is not considered in any of the available approaches.

Concerning the representation and classification of semantic sensor metadata, there is an increasing interest in using ontologies for representing sensor data, enabling interoperability, discovery and publication of sensor data on the web. However, the published sensor metadata usually lacks the necessary information to be of use for a wide community of users. Characterizing and classifying the types of sensor data sources is a problem that has been tackled in the time series mining and analysis community but has been seldom explored for semantic metadata management.

As we stated in Chapter 3 our work presents several contributions to these research

problems. We review them below, in the light of the evidence showed throughout the previous chapters and the evaluations performed.

## 9.1 Review of Contributions

The main contribution of our work is focused on the first problem, and consists of a **novel approach for accessing and querying existing streaming data sources**, through the query rewriting semantics of the proposed SPARQL<sub>Stream</sub> language. In particular, we have expanded over the work on RDB2RDF techniques for the case of streaming data, adding the notions of continuous queries, streamed evaluation and time-based windows. In particular, we can detail concrete contributions associated to this research problem as follows:

- **We defined the SPARQL<sub>Stream</sub> language, its syntax and semantics** in Chapter 4. SPARQL<sub>Stream</sub> has been designed considering it as an extension to SPARQL 1.1 with window operators as the ones present in DSMS and CEP languages. Unlike other SPARQL extensions for streaming data, SPARQL<sub>Stream</sub> is oriented to reuse existing streaming query engines, thus providing an ontological view over these sources. Our objective with this approach is to hide the heterogeneity of data schemas of different sensor deployments by using a common model (e.g. an ontology network with the SSN ontology at the core).
- **We proposed a query rewriting approach for expressing SPARQL<sub>Stream</sub> queries as relational streaming algebra expressions.** The formalization of the query rewriting semantics of SPARQL<sub>Stream</sub> is fully described in Chapter 4 and is based on the existence of mappings from stream schemas to RDF terms. This novel approach departs from fixed-schema transformations of previous works, and explores for the first time the possibility of applying and extending concepts of the RDB2RDF and ontology-based data access areas to data streams.
- **We enabled the query rewriting and data translation of query results** from relational-based streams and event streams using our query rewriting approach (Chapter 5). The abstract representation of SPARQL<sub>Stream</sub> queries is essentially relational, and we provided evidence of its applicability for streaming relational and event models. These are the data and query models currently exploited in the research community and the industry, as described in Section 2.1, and we are experiencing an increasing demand for applications that make use of them. In this

context, the SPARQL<sub>Stream</sub> query rewriting approach provides flexibility to be used in conjunction with very different streaming data schemas and models.

- We provided an **implementation of a SPARQL<sub>Stream</sub> continuous query engine**, pluggable to a wide range of underlying streaming data engines. We implemented the proposed approach such that it can reuse one of several existing stream management systems, complex event processors, or even sensor middleware. Our approach has been implemented using four stream and sensor management systems, each one with different goals and querying capabilities, showing that these principles can be applied to a potentially wide range of situations. We have shown concrete examples of real-world use cases where this technology can be applied (Chapter 7).
- We proposed and implemented the definition of **mappings from streaming data schemas to ontologies using the R2RML W3C Recommendation**, extending its use for streams for the first time (Chapter 5). The usage of R2RML mappings allows a declarative definition of how the virtual RDF graphs are related to real streaming data sources, and provides a reusable abstraction that can be applied in many scenarios as seen in Chapter 7. Moreover, the well-founded semantics of the query rewriting mechanism of SPARQL<sub>Stream</sub>, has been shown to fit with this type of mappings in Chapter 5.
- We have provided an implementation that has been thoroughly evaluated, for both pull and push based delivery modes, especially for assessing the potential overhead of query rewriting and data translation to the overall query evaluation process (Chapter 8).

While these contributions targeted the problem of streaming data querying, and the use of ontology terms for this purpose, we also covered the second research problem, referred to metadata management for sensors, which are one of the main sources of data streams. We addressed the challenges of classifying and deriving semantic metadata, and we can summarize our contributions as follows:

- We **defined a representation of sensor time series that captures slopes information** that is useful to characterize types of sensor data (Chapter 6). In particular we evidenced that this representation helps discerning from the different observed properties of sensor measurements, which are potentially unknown at web scale.

- Using the slopes representation, we **devised a method for classifying sensor time series and determining the type of data**, using a classification scheme (Chapter 6). This approach assumes a basis of existing already classified time series, and has been evaluated with real world datasets. We have also compared this approach with other alternative classification techniques in Chapter 8.
- We used the classification method to **derive sensor metadata observed properties**, so that they can be integrated with the semantic sensor metadata and later be used in queries. Using this approach we allow determining observed properties of sensor datasets, so that they are easily discoverable and can be properly queried, using semantic-aware languages such as SPARQL<sub>Stream</sub>.

## 9.2 Future Work

The work presented in this thesis provides contributions that enable querying and accessing semantic sensor observations and metadata on the Web, and constitute a step forward in this area. However, we have already identified open issues and related work to follow up, which in some cases we have started to explore.

Our work provides substantial evidence that we can apply ontology-based query processing for streaming data sources. The theoretical foundations of query rewriting and usage of mappings allow using external event or relational streaming engines. We have shown the application of these principles with concrete implementations such as Esper, SNEE or middleware including Cosm and GSN. Nevertheless, we have seen in Section 8.2 some of the functionalities available in event processors are not covered by SPARQL<sub>Stream</sub>. We plan to analyze the addition of such features, and include them if it is feasible. One of the advantages of our approach is that it is flexible enough to incorporate different types of underlying streaming engines, provided that the queries are rewritable. Therefore if we add an operator in SPARQL<sub>Stream</sub> that has an equivalence in the target language (e.g. Esper), then it can be easily added.

Along these lines, we have also seen in Chapter 8 that the available prototype of the SPARQL<sub>Stream</sub> evaluator needs to implement the complete set of features provided by the language. We realized that none of the existing approaches that target RDF stream processing covers very relevant features of SPARQL 1.1 and reasoning. The complementary work of SRBench points at this direction, and is not intended just for internal use but to

be exploited by the community. In this context, we are also starting to address the need for standardizing this type of query language extensions, for instance at the W3C level.

Concerning the support of mixed static and streaming data sources, the current SPARQL<sub>Stream</sub> implementation assumes that the query engine can handle them appropriately. For instance most DSMS support both tables and streams, and a query semantics that allow joins between them. Nonetheless, in the context of Linked Data or other types of Web data publishing, static data is often remote or accessible through federated query endpoints. Integrating this type of functionality to SPARQL<sub>Stream</sub> is an interesting ongoing work that we are starting to tackle.

Regarding the sensor metadata characterization, our approach can be applied an expanded in many ways. First of all, it was evaluated within the same dataset, and in the future we will study its applicability in an inter-dataset classification. This framework could be used in the future for other tasks such as clustering, or for identifying simple patterns in streams of sensor data. Moreover, complex symbolizations consisting of sequences of slopes could be considered, which would represent more complete patterns that can be exploited. Also, we can consider building a more complex representation that includes not only the slopes information but also the value ranges, and even tags and labels provided the data publishers. This may enable a more complete and accurate extraction of metadata that enriches the growing Semantic Sensor Web.

## 9.3 Open Research Problems

The body of our work addressed open problems in the area of semantic sensor networks. These advances opened the way for exploring other related research problems that we mention briefly in this section.

**Publishing Linked Stream Data.** Our approach for querying streaming data requires a SPARQL<sub>Stream</sub>-enabled endpoint. This supposes that all streaming RDF graphs and triples are virtual in nature, not stored as in traditional RDF datasets. Event though all the virtual triples that are queried have a URI, in the current settings they are not necessarily dereferenceable. This breaks the normal conception of Web publishing standards such as Linked Data, but this is due to a lack of support of these technologies for data streams. In fact, we most likely do not need to store all observations but only



summarizations, and expose the most recent ones in SPARQL<sub>Stream</sub>-like endpoints. We think that there is a great potential for research work in this area, and although there have been some proposals in these lines (e.g. (Le-Phuoc et al., 2011b; Wei and Barnaghi, 2009)), there are still many uncovered use cases to work on.

**Parallel Stream Processing.** Massive online and batch processing of data is a current hot research area, that is specially important for data streams. Parallelization of stream processing is an area that is starting to gain wide adoption in financial market analysis, social networks or traffic monitoring. Several projects emerged to cover these issues, including S4<sup>1</sup> or Storm<sup>2</sup>. However, all these efforts suffer from similar problems than those of DSMS with respect to heterogeneity and interoperability. We believe that some of the ideas developed in this thesis could be fruitful in that area, although there are several technical and scientific issues to take into account.

**Query federation.** Data streams coming from sensor networks or mobile devices are distributed in nature. However, applications require aggregating, summarizing or comparing streaming data from these different sources in real time. Federated query processing and distributed evaluation of queries has already been addressed by the sensor networks community (e.g. (Galpin et al., 2011; Madden et al., 2005)) and also in static RDF query processing (e.g. (Acosta et al., 2011; Buil-Aranda et al., 2011)) but it is still largely unexplored in the area of streaming SPARQL evaluation. We envision an ecosystem of sensing and actuating devices that may expose their data directly through Web-standard interfaces for direct consumption or federation, or through proprietary formats to query engines equivalent to the ones SPARQL<sub>Stream</sub> proposes. And on top of these heterogeneous systems, we could investigate mediator-based query optimization techniques for data integration.

**Sensor data workflows.** In scientific data analysis, it is more and more common to rely not only on large volumes of datasets, but also on highly dynamic streaming data sources. These data streams may be part of complex workflows that describe a data transformation and analysis task, which is potentially publishable and re-playable. Sharing and annotating these workflows has been tackled by researchers for a long time, but this work can also be extended for streaming data, which poses the problems of dynamicity, reproducibility of dynamic streaming data, and provenance.

---

<sup>1</sup><http://incubator.apache.org/s4/>

<sup>2</sup><http://storm-project.net/>

**Stream Reasoning.** In the field of reasoning, streaming data has only recently been considered (Della Valle et al., 2009) and so far there is limited support in terms of query processing. The usage of query rewriting techniques that consider the ontologies to produce expanded queries to a database has been extensively studied in the latest years, exclusively for static data (Calvanese et al., 2005; Pérez-Urbina et al., 2009). Approaches such as SPARQL<sub>Stream</sub> that are based on ontology-based data access techniques can directly benefit from the query expansion reasoning methods, although there is also a need for providing clear theoretical foundations to explain the scope and expressiveness that can be achieved.

**Sensor pattern classification.** We have studied a particular type of classification problem, oriented towards the identification of observed properties of sensor data. Very similar techniques can be used for other purposes which can be broadly described as pattern classification. Although this type of task has been widely studied by time series analysis scientists, we see a need for combining it with streaming query processing, especially on live data streams. SPARQL<sub>Stream</sub> and other query engines for graph-based data could be empowered with these data analysis features to exploit pattern-based classification on continuous queries, going beyond simple filtering or aggregation operators.

**Statistical and quality analysis.** Sensor data and its corresponding metadata are consumed by specialized users that perform analysis over it, and usually require common statistical models and tools that are applied over and over for each dataset they need to process. These tasks can be integrated in a processing stack that includes data acquisition and querying. All these steps could be represented using the previously mentioned workflow-based concepts, but we would require mappings from sensor-observation models to statistical models that are well understood by scientific analysts. Furthermore, we see the need for enabling data quality filtering and analysis for this type of data streams, as they usually come from noisy and partially unreliable sources.

All in all, we have stepped into a research area where the challenges demand new solutions and proposals that bring together different fields, ranging from data stream processing, sensor data analysis, ontology engineering Web publishing, reasoning, and many more.



# References

- Abadi, D. J., Ahmad, Y., Balazinska, M., Cetintemel, U., Cherniack, M., Hwang, J.-H., Lindner, W., Maskey, A. S., Rasin, A., Ryvkina, E., Tatbul, N., Xing, Y., and Zdonik, S. (2005). The Design of the Borealis Stream Processing Engine. In *Proc. 2nd Conference on Innovative Database Research CIDR 2005*, pages 277–289. [www.cidrdb.org](http://www.cidrdb.org).
- Abadi, D. J., Carney, D., Cetintemel, U., Cherniack, M., Convey, C., Lee, S., Stonebraker, M., Tatbul, N., and Zdonik, S. (2003). Aurora: a new model and architecture for data stream management. *The VLDB Journal*, 12(2):120–139.
- Abadi, D. J., Lindner, W., Madden, S., and Schuler, J. (2004). An integration framework for sensor networks and data stream management systems. In *Proc. 30th international Conference on Very Large Data Bases VLDB 2004*, pages 1361–1364. VLDB Endowment.
- Aberer, K., Hauswirth, M., and Salehi, A. (2006). A middleware for fast and flexible sensor network deployment. In *Proc. 32nd International Conference on Very Large Data Bases VLDB 2006*, pages 1199–1202. VLDB Endowment.
- Abiteboul, S., Hull, R., and Vianu, V. (1995). *Foundations of Databases*. Addison-Wesley.
- Acosta, M., Vidal, M., Lampo, T., Castillo, J., and Ruckhaus, E. (2011). Anapsid: An adaptive query processing engine for SPARQL endpoints. In *Proc. 10th International Semantic Web Conference ISWC 2011*, pages 18–34. Springer.
- Agrawal, R., Faloutsos, C., and Swami, A. (1993). Efficient similarity search in sequence databases. *Foundations of Data Organization and Algorithms*, 730:69–84.
- Anicic, D., Fodor, P., Rudolph, S., and Stojanovic, N. (2011). Ep-sparql: a unified language for event process-

- ing and stream reasoning. In *Proc. 20th international conference on World wide web WWW 2011*, pages 635–644. ACM.
- Arasu, A., Babcock, B., Babu, S., Cieslewicz, J., Datar, M., Ito, K., Motwani, R., Srivastava, U., and Widom, J. (2007). STREAM: The Stanford data stream management system. In Garofalakis, M., Gehrke, J., and Rastogi, R., editors, *Data Stream Management: Processing High-Speed Data Streams*. Springer-Verlag.
- Arasu, A., Babcock, B., Babu, S., McAlister, J., and Widom, J. (2004a). Characterizing memory requirements for queries over continuous data streams. *ACM Transactions on Database Systems*, 29(1):162–194.
- Arasu, A., Babu, S., and Widom, J. (2006). The CQL continuous query language: semantic foundations and query execution. *The VLDB Journal*, 15(2):121–142.
- Arasu, A., Cherniack, M., Galvez, E., Maier, D., Maskey, A., Ryvkina, E., Stonebraker, M., and Tibbetts, R. (2004b). Linear road: a stream data management benchmark. In *Proceedings of the Thirtieth international conference on Very large data bases-Volume 30*, pages 480–491. VLDB Endowment.
- Arenas, M., Conca, S., and Pérez, J. (2012). Counting beyond a yottabyte, or how sparql 1.1 property paths will prevent adoption of the standard. In *Proceedings of the 21st international conference on World Wide Web*, pages 629–638. ACM.
- Arens, Y., Chee, C. Y., Hsu, C.-N., and Knoblock, C. A. (1993). Retrieving and integrating data from multiple information sources. *International Journal of Intelligent and Cooperative Information Systems*, 2(2):127–158.
- Auer, S., Dietzold, S., Lehmann, J., Hellmann, S., and Aumüller, D. (2009). Triplify: light-weight linked data publication from relational databases. In *Proc. 18th international conference on World wide web WWW 2009, WWW '09*, pages 621–630, New York, NY, USA. ACM.
- Avancha, S., Patel, C., and Joshi, A. (2004). Ontology-driven adaptive sensor networks. In *Proc. 1st International Conference on Mobile and Ubiquitous Systems, Networking and Services Mobiquitous 2004*, pages 194–202. IEEE Computer Society.
- Babcock, B., Babu, S., Datar, M., Motwani, R., and Widom, J. (2002). Models and issues in data stream systems. In *Proc. 21st ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems PODS 2002*, pages 1–16. ACM.

## References

---

- Babitski, G., Bergweiler, S., Hoffmann, J., Schön, D., Stasch, C., and Walkowski, A. (2009). Ontology-based integration of sensor web services in disaster management. In *Proc. 3rd International Conference on GeoSpatial Semantics GeoS 2009*, pages 103–121. Springer.
- Babu, S. and Widom, J. (2001). Continuous queries over data streams. *ACM SIGMOD Record*, 30(3):109–120.
- Bai, Y., Thakkar, H., Wang, H., Luo, C., and Zaniolo, C. (2006). A data stream language and system designed for power and extensibility. In *Proc. 15th ACM international conference on Information and knowledge management CIKM 2006*, pages 337–346, New York, NY, USA. ACM.
- Barbieri, D. and Della Valle, E. (2010). A Proposal for Publishing Data Streams as Linked Data – a position paper. In *Proc. WWW2010 Workshop on Linked Data on the Web LDOW 2010*, volume 628. CEUR-WS.org.
- Barbieri, D. F., Braga, D., Ceri, S., Della Valle, E., and Grossniklaus, M. (2010a). C-SPARQL: A continuous query language for RDF data streams. *International Journal of Semantic Computing*, 4(1):3–25.
- Barbieri, D. F., Braga, D., Ceri, S., and Grossniklaus, M. (2010b). An execution environment for C-SPARQL queries. In *Proc. 13th International Conference on Extending Database Technology EDBT 2010*, pages 441–452, Lausanne, Switzerland. ACM.
- Barga, R. S., Goldstein, J., Ali, M. H., and Hong, M. (2007). Consistent streaming through time: A vision for event stream processing. In *Proc. 3rd Conference on Innovative Data Systems Research CIDR 2007*, pages 363–374. www.cidrdb.org.
- Barnaghi, P., Ganz, F., Henson, C., and Sheth, A. (2012). Computing perception from sensor data. In *Proc. 2012 IEEE Sensors Conference*. (to appear).
- Barnaghi, P., Meissner, S., Presser, M., and Moessner, K. (2009). Sense and sensability: Semantic data modelling for sensor networks. In *Conference Proceedings of the ICT Mobile Summit*. IIMC.
- Barrasa, J., Corcho, O., and Gómez-Pérez, A. (2004). R2O, an extensible and semantically based database-to-ontology mapping language. In *Proc. 2nd Workshop on Semantic Web and Databases SWDB 2004*, pages 1069–1070. Springer.
- Beneventano, D., Bergamaschi, S., Vincini, M., Orsini, M., and Nana Rodriguez, C. (2007). Query translation in heterogeneous sources in momis data transformation systems. In *Proc. 3rd International Workshop on Database Interoperability InterDB 2007 at VLDB*.

- Berners-Lee, T. (2006). Linked data - design issues <http://www.w3.org/DesignIssues/LinkedData.html>. Technical report, W3C.
- Bizer, C. and Cyganiak, R. (2007). D2RQ . Lessons Learned. W3C Workshop on RDF Access to Relational Databases.
- Bizer, C. and Schultz, A. (2009). The Berlin SPARQL benchmark. *International Journal on Semantic Web and Information Systems (IJSWIS)*, 5(2):1–24.
- Bolles, A., Grawunder, M., and Jacobi, J. (2008). Streaming SPARQL - extending SPARQL to process data streams. In *Proc. 5th Extended Semantic Web Conference ESWC 2008*, pages 448–462. Springer-Verlag.
- Bonnet, P., Gehrke, J., and Seshadri, P. (2001). Towards sensor database systems. In *Proc. 2nd International Conference on Mobile Data Management MDM 2001*, pages 3–14. Springer.
- Botts, M., Percivall, G., Reed, C., and Davidson, J. (2006). Ogc® sensor web enablement: Overview and high level architecture. In *Proc. 2nd International Conference on GeoSensor Networks GSN 2006*, volume 4540, pages 175–190. Springer.
- Brenna, L., Demers, A., Gehrke, J., Hong, M., Ossher, J., Panda, B., Riedewald, M., Thatte, M., and White, W. (2007). Cayuga: a high-performance event processing engine. In *Proc. ACM SIGMOD International Conference on Management of Data*, pages 1100–1102. ACM.
- Brenninkmeijer, C. Y., Galpin, I., Fernandes, A. A., and Paton, N. W. (2008). A semantics for a query language over sensors, streams and relations. In *Proc. 25th British National Conference on Databases BNCOD 2008*, pages 87–99. Springer-Verlag.
- Bröring, A., Janowicz, K., Stasch, C., and Kuhn, W. (2009). Semantic challenges for sensor plug and play. In *Proc. 9th International Symposium on Web and Wireless Geographical Information Systems*, volume 5886, pages 72–86. Springer.
- Buil-Aranda, C., Arenas, M., and Corcho, O. (2011). Semantics and optimization of the SPARQL 1.1 federation extension. In *Proc. 8th Extended Semantic Web Conference ESWC 2011*, pages 1–15. Springer.
- Buragohain, C., Shrivastava, N., and Suri, S. (2007). Space efficient streaming algorithms for the maximum error histogram. In *Proc. IEEE 23rd International Conference on Data Engineering ICDE 2007*, pages 1026–1035. IEEE.

## References

---

- Calbimonte, J.-P., Corcho, O., and Gray, A. (2010a). Ontology-based access to streaming data. In *Poster Proc. 7th Extended Semantic Web Conference ESWC 2010*.
- Calbimonte, J.-P., Corcho, O., and Gray, A. J. G. (2010b). Enabling ontology-based access to streaming data sources. In *Proc. 9th International Semantic Web Conference ISWC 2010*, pages 96–111.
- Calbimonte, J.-P., Corcho, O., and Gray, A. J. G. (2011a). Implementation and deployment of the SemSorGrid4Env ontology-based data integration service, Phase II. Deliverable D4.2v2. Technical report, SemSorGrid4Env FP7.
- Calbimonte, J.-P., García-Castro, R., Corcho, O., and Rodríguez, J. (2011b). Evaluation of the ontology-based data integration service and the ontologies. Deliverable D4.4. Technical report, SemSorGrid4Env FP7.
- Calbimonte, J.-P., Jeung, H., and Corcho, O. (2011c). Querying semantically enriched sensor observations: Short paper. In *Proc. 6th International Workshop on Semantic Business Process Management SBPM 2011 at ESWC*.
- Calbimonte, J.-P., Jeung, H., Corcho, O., and Aberer, K. (2011d). Semantic sensor data search in a large-scale federated sensor network. In *Proc. 4th International Workshop on Semantic Sensor Networks SSN2011 at ISWC*, pages 14–29. CEUR-WS.org.
- Calbimonte, J.-P., Jeung, H., Corcho, O., and Aberer, K. (2012a). Enabling query technologies for the semantic sensor web. *International Journal On Semantic Web and Information Systems (IJSWIS)*, 8(1):43–63.
- Calbimonte, J.-P., Yan, Z., Jeung, H., Corcho, O., and Aberer, K. (2012b). Deriving semantic sensor metadata from raw measurements. In *Proc. 5th International Workshop on Semantic Sensor Networks SSN2012 at ISWC*, volume 904, pages 33–48. CEUR-WS.org.
- Calì, A., Calvanese, D., Giacomo, G. D., Lenzerini, M., Naggar, P., and Vernacotola, F. (2003a). Ibis: Semantic data integration at work. In *Proc. 15th international conference on Advanced information systems engineering CAISE 2003*, pages 79–94. Springer-Verlag.
- Calì, A., De Nigris, S., Lembo, D., Messineo, G., Rosati, R., and Ruzzi, M. (2003b). Dis@dis: A system for semantic data integration under integrity constraints. In *Proc. 4th International Conference on Web Information Systems Engineering WISE 2003*, page 335, Washington, DC, USA. IEEE Computer Society.



- Calvanese, D., De Giacomo, G., Lembo, D., Lenzerini, M., and Rosati, R. (2005). DL-Lite: Tractable description logics for ontologies. In *Proc. 20th National Conference on Artificial Intelligence AAAI 2005*, volume 2, pages 602–607. AAAI Press.
- Calvanese, D., De Giacomo, G., Lenzerini, M., Nardi, D., and Rosati, R. (1998). Description logic framework for information integration. In *Proc. 6th International Conference on the Principles of Knowledge Representation and Reasoning KR'98*, pages 2–13. Morgan Kaufmann.
- Cerbah, F. (2008). Learning highly structured semantic repositories from relational databases the RDBToOnto tool. In *Proc. 5th Extended Semantic Web Conference ESWC 2008*, pages 777–781. Springer.
- Chakrabarti, K., Keogh, E., Mehrotra, S., and Pazzani, M. (2002). Locally adaptive dimensionality reduction for indexing large time series databases. *ACM Transactions on Database Systems*, 27(2):188–228.
- Chandrasekaran, S., Cooper, O., Deshpande, A., Franklin, M. J., Hellerstein, J. M., Hong, W., Krishnamurthy, S., Madden, S. R., Reiss, F., and Shah, M. A. (2003). TelegraphCQ: continuous dataflow processing for an uncertain world. In *Proc. ACM SIGMOD International Conference on Management of Data*, pages 668–668. ACM.
- Chandrasekaran, S. and Franklin, M. J. (2003). PSoup: a system for streaming queries over streaming data. *The VLDB Journal*, 12(2):140–156.
- Chandy, M. K., Etzion, O., and von Ammon, R. (2011). The event processing manifesto. In Chandy, K. M., Etzion, O., and von Ammon, R., editors, *Event Processing*, number 10201 in Dagstuhl Seminar Proceedings, Dagstuhl, Germany. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany.
- Chebotko, A., Lu, S., and Fotouhi, F. (2009). Semantics preserving SPARQL-to-SQL translation. *Data & Knowledge Engineering*, 68(10):973–1000.
- Chen, J., DeWitt, D. J., Tian, F., and Wang, Y. (2000). NiagaraCQ: a scalable continuous query system for internet databases. *ACM SIGMOD Record*, 29(2):379–390.
- Compton, M., Barnaghi, P., Bermudez, L., García-Castro, R., Corcho, O., Cox, S., Graybeal, J., Hauswirth, M., Henson, C., Herzog, A., Huang, V., Janowicz, K., Kelsey, W. D., Phuoc, D. L., Lefort, L., Leggieri, M., Neuhaus, H., Nikolov, A., Page, K., Passant, A., Sheth, A., and Taylor, K. (2012). The SSN ontology of the W3C semantic sensor network incubator group. *Journal of Web Semantics*, 17:25–32.

## References

---

- Compton, M., Henson, C., Lefort, L., Neuhaus, H., and Sheth, A. (2009a). A survey of the semantic specification of sensors. In *Proc. 2nd International Workshop on Semantic Sensor Networks SSN 2009 at ISWC*, pages 17–32. CEUR-WS.org.
- Compton, M., Neuhaus, H., Taylor, K., and Tran, K. (2009b). Reasoning about sensors and compositions. In *Proc. 2nd International Workshop on Semantic Sensor Networks SSN 2009 at ISWC*, pages 33–48. CEUR-WS.org.
- Corcho, O. and García-Castro, R. (2010). Five challenges for the Semantic Sensor Web. *Semantic Web*, 1(1):121–125.
- Corcho, O., Priyatna, F., Fortuna, C., Grobelnik, M., Calbimonte, J.-P., García-Silva, A., Jeung, H., Novak, B., and Moraru, A. (2011). Characterisation mechanisms for unknown data sources. Deliverable D1.1. Technical report, PlanetData FP7.
- Cranor, C., Johnson, T., Spataschek, O., and Shkapenyuk, V. (2003). Gigascope: a stream database for network applications. In *Proc. ACM SIGMOD International Conference on Management of Data*, pages 647–651, New York, NY, USA. ACM.
- Cristianini, N. and Shawe-Taylor, J. (2000). *An introduction to support vector machines and other kernel-based learning methods*. Cambridge University Press.
- Cugola, G. and Margara, A. (2010). Tesla: a formally defined event specification language. In *Proc. 4th ACM International Conference on Distributed Event-Based Systems DEBS 2010*, pages 50–61. ACM.
- Cugola, G. and Margara, A. (2011). Processing flows of information: From data stream to complex event processing. *ACM Computing Surveys*, 44(3):15:1–15:62.
- De Mel, G., Pham, T., Damarla, T., Vasconcelos, W., and Norman, T. (2011). Semantically enriched data for effective sensor data fusion. In *SPIE Defense, Security, and Sensing*, volume 8047. SPIE Proceedings.
- Deligiannakis, A., Kotidis, Y., and Roussopoulos, N. (2004). Compressing historical information in sensor networks. In *Proc. ACM SIGMOD international Conference on Management of data*, pages 527–538, New York, NY, USA. ACM.
- Della Valle, E., Ceri, S., Braga, D., Celino, I., Fensel, D., van Harmelen, F., and Unel, G. (2009). Research chapters in the area of stream reasoning. In *Proc. 1st International Conference on Stream Reasoning SR 2009*, pages 1–9. CEUR-WS.org.

- Ding, H., Trajcevski, G., Scheuermann, P., Wang, X., and Keogh, E. J. (2008). Querying and mining of time series data: experimental comparison of representations and distance measures. *Journal Proceedings of the VLDB Endowment*, 1(2):1542–1552.
- Doan, A. and Halevy, A. Y. (2005). Semantic integration research in the database community: A brief survey. *AI Magazine*, 26(1):83–94.
- Duan, S., Kementsietsidis, A., Srinivas, K., and Udrea, O. (2011). Apples and oranges: a comparison of rdf benchmarks and real rdf datasets. In *Proceedings of the 2011 international conference on Management of data*, pages 145–156. ACM.
- Eid, M., Liscano, R., and El Saddik, A. (2007). A universal ontology for sensor networks data. In *Proc. IEEE International Conference on Computational Intelligence for Measurement Systems and Applications CIMSAS 2007*, pages 59–62. IEEE.
- Erling, O. and Mikhailov, I. (2007). RDF support in the Virtuoso DBMS. In *Proc. 1st Conference on Social Semantic Web CSSW 2007*, volume 113 of *LNI*, pages 59–68.
- Evensen, P. and Meling, H. (2009). Sensewrap: A service oriented middleware with sensor virtualization and self-configuration. In *Proc. 5th International Conference on Intelligent Sensors, Sensor Networks and Information Processing ISSNIP 2009*, pages 261–266. IEEE.
- Faloutsos, C., Ranganathan, M., and Manolopoulos, Y. (1994). Fast subsequence matching in time-series databases. In *Proc. ACM SIGMOD International conference on Management of data*, pages 419–429, New York, NY, USA. ACM.
- Friedman, M., Levy, A., and Millstein, T. (1999). Navigational plans for data integration. In *Proc. 16th National conference on Artificial intelligence AAAI 1999*, pages 67–73, Menlo Park, CA, USA. AAAI Press.
- Gaber, M., Zaslavsky, A., and Krishnaswamy, S. (2005). Mining data streams: a review. *ACM SIGMOD Record*, 34(2):18–26.
- Galpin, I., Brenninkmeijer, C., Gray, A., Jabeen, F., Fernandes, A., and Paton, N. (2011). SNEE: a query processor for wireless sensor networks. *Distributed and Parallel Databases*, 29(1):31–85.
- Galpin, I., Brenninkmeijer, C. Y., Jabeen, F., Fernandes, A. A., and Paton, N. W. (2008). An architecture for query optimization in sensor networks. In *Proc. IEEE 24th International Conference on Data Engineering ICDE 2008*, pages 1439–1441. IEEE.

## References

---

- Galpin, I., Brenninkmeijer, C. Y., Jabeen, F., Fernandes, A. A., and Paton, N. W. (2009). Comprehensive optimization of declarative sensor network queries. In *Proc. 21st International Conference on Scientific and Statistical Database Management SSDBM 2009*, pages 339–360. Springer.
- Gandhi, S., Nath, S., Suri, S., and Liu, J. (2009). Gamps: compressing multi sensor data by grouping and amplitude scaling. In *Proc. ACM SIGMOD International Conference on Management of data*, pages 771–784, New York, NY, USA. ACM.
- Garcia-Molina, H., Papakonstantinou, Y., Quass, D., Sagiv, Y., Ullman, J., Vassalos, V., and Widom, J. (1997). The TSIMMIS approach to mediation: Data models and languages. *Journal of Intelligent Information Systems*, 8:117–132.
- Gay, D., Levis, P., Von Behren, R., Welsh, M., Brewer, E., and Culler, D. (2003). The nesC language: A holistic approach to networked embedded systems. In *Proc. ACM SIGPLAN conference on Programming language design and implementation PLDI 2003*, volume 38, pages 1–11. ACM.
- Gehrke, J., Korn, F., and Srivastava, D. (2001). On computing correlated aggregates over continual data streams. *ACM SIGMOD Record*, 30(2):13–24.
- Genesereth, M. R., Keller, A. M., and Duschka, O. M. (1997). Infomaster: an information integration system. *ACM SIGMOD Record*, 26(2):539–542.
- Geurts, P. (2001). Pattern extraction for time series classification. In *Proc. 5th European Conference on Principles of Data Mining and Knowledge Discovery PKDD 2001*, pages 115–127. Springer-Verlag.
- Gibbons, P., Karp, B., Ke, Y., Nath, S., and Seshan, S. (2003). Irisnet: An architecture for a worldwide sensor web. *IEEE Pervasive Computing*, 2(4):22–33.
- Goasdoué, F., Lattès, V., and Rousset, M.-C. (2000). The use of CARIN language and algorithms for information integration: The PICSEL system. *International Journal of Cooperative Information Systems*, 9(4):383–401.
- Golab, L. and Özsu, M. T. (2003). Issues in data stream management. *ACM SIGMOD Record*, 32(2):5–14.
- Golab, L. and Özsu, M. T. (2003). Processing sliding window multi-joins in continuous queries over data streams. In *Proc. 29th international conference on Very large data bases VLDB 2003, VLDB '03*, pages 500–511. VLDB Endowment.

- González-Valenzuela, S., Chen, M., and Leung, V. (2011). Mobility support for health monitoring at home using wearable sensors. *IEEE Transactions on Information Technology in Biomedicine*, 15(4):539–549.
- Gray, A., García-Castro, R., Kyzirakos, K., Karpathiotakis, M., Calbimonte, J., Page, K., Sadler, J., Frazer, A., Galpin, I., Fernandes, A., et al. (2011a). A semantically enabled service architecture for mashups over streaming and stored data. In *Proc. 8th Extended Semantic Web Conference ESWC 2011*, pages 300–314. Springer.
- Gray, A., Sadler, J., Kit, O., Kyzirakos, K., Karpathiotakis, M., Calbimonte, J., Page, K., García-Castro, R., Frazer, A., and Galpin, I. e. a. (2011b). A semantic sensor web for environmental decision support applications. *Sensors*, 11(9):8855–8887.
- Gray, A. J. G., Galpin, I., Fernandes, A. A. A., and Paton, N. W. (2009). Web services data access and integration - the data stream realisation (WS-DAIStreaming) specification. Technical report, University of Manchester.
- Gray, A. J. G., Galpin, I., Fernandes, A. A. A., Paton, N. W., Page, K., Sadler, J., Kyzirakos, K., Koubarakis, M., Calbimonte, J.-P., Corcho, O., García-Castro, R., Gabaldón, J., and Aparicio, J. (2010). SemSor-Grid4Env Architecture - Phase II. Deliverable D1.3v2. Technical report, SemSorGrid4Env FP7.
- Gulisano, V., Jimenez-Peris, R., Patino-Martinez, M., and Valduriez, P. (2010). Streamcloud: A large scale data streaming system. In *Proc. IEEE 30th International Conference on Distributed Computing Systems ICDCS 2010*, pages 126–137. IEEE.
- Guo, Y., Pan, Z., and Heflin, J. (2005). Lubm: A benchmark for owl knowledge base systems. *Web Semantics: Science, Services and Agents on the World Wide Web*, 3(2):158–182.
- Gurgen, L., Roncancio, C., Labbé, C., Bottaro, A., and Olive, V. (2008). SStreaMWare: a service oriented middleware for heterogeneous sensor data management. In *Proc. 5th International conference on Pervasive services ICPS 2008*, pages 121–130, New York, NY, USA. ACM.
- Gutierrez, C., Hurtado, C., and Vaisman, A. (2007). Introducing time into RDF. *IEEE Transactions on Knowledge and Data Engineering*, 19(2):207–218.
- Halevy, A. Y. (2001). Answering queries using views: A survey. *The VLDB Journal*, 10(4):270–294.
- Henson, C., Pschorr, J., Sheth, A., and Thirunarayan, K. (2009). SemSOS: Semantic Sensor Observation Service. In *Proc. 2009 International Symposium on Collaborative Technologies and Systems CTS 2009*, pages 44–53. IEEE.

## References

---

- Hey, A., Tansley, S., and Tolle, K. (2009). *The fourth paradigm: data-intensive scientific discovery*. Microsoft Research Redmond, WA.
- Huang, V. and Javed, M. (2008). Semantic sensor information description and processing. In *Proc. 2nd International Conference on Sensor Technologies and Applications SENSORCOMM 2008*, pages 456–461. IEEE.
- Huhns, M., Jacobs, N., Ksiezyk, T., Shen, W.-M., Singh, M., and Cannata, P. (1993). Integrating enterprise information models in carnot. In *Proc. International Conference on Intelligent and Cooperative Information Systems*, pages 32–42. IEEE.
- Ibrahim, I. K., Kronsteiner, R., and Kotsis, G. (2005). A semantic solution for data integration in mixed sensor networks. *Computer Communications*, 28(13):1564–1574.
- Jagadish, H. V., Mumick, I. S., and Silberschatz, A. (1995). View maintenance issues for the chronicle data model (extended abstract). In *Proc. 14th ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems PODS'95*, pages 113–124, New York, NY, USA. ACM.
- Janowicz, K. and Compton, M. (2010). The Stimulus-Sensor-Observation Ontology Design Pattern and its Integration into the Semantic Sensor Network Ontology. In *Proc. 3rd International Workshop on Semantic Sensor Networks SSN 2010 at ISWC*, pages 7–11. CEUR-WS.org.
- Jeung, H., Sarni, S., Paparrizos, I., Sathe, S., Aberer, K., Dawes, N., Papaioannou, T., and Lehning, M. (2010). Effective Metadata Management in Federated Sensor Networks. In *Proc. 3rd International Conference on Sensor Networks, Ubiquitous, and Trustworthy Computing SUTC 2010*, pages 107–114. IEEE.
- Karpathiotakis, M., Kyzirakos, K., Kaoudi, Z., Fisikopoulos, V., Miliaraki, I., Koubarakis, M., Ioannidis, Y., and Hatzopoulos, M. (2011). Implementation and deployment of the registry services - Phase II. Deliverable D3.3v2. Technical report, SemSorGrid4Env FP7.
- Kasetty, S., Stafford, C., Walker, G., Wang, X., and Keogh, E. (2008). Real-time classification of streaming sensor data. In *Proc. 20th IEEE International Conference on Tools with Artificial Intelligence ICTAI'08*, volume 1, pages 149–156. IEEE.
- Keogh, E., Chakrabarti, K., Pazzani, M., and Mehrotra, S. (2001). Dimensionality reduction for fast similarity search in large time series databases. *Knowledge and information Systems*, 3(3):263–286.

- Keogh, E., Chu, S., Hart, D., and Pazzani, M. (2004). Segmenting time series: A survey and novel approach. *Data mining in time series databases*, 57.
- Keogh, E. and Kasetty, S. (2003). On the need for time series data mining benchmarks: a survey and empirical demonstration. *Data Mining and Knowledge Discovery*, 7(4):349–371.
- Keogh, E. and Smyth, P. (1997). A probabilistic approach to fast pattern matching in time series databases. In *Proc. 3rd International Conference on Knowledge Discovery and Data Mining KDD'97*, pages 24–30. AAAI Press.
- Kim, J., Kwon, H., Kim, D., Kwak, H., and Lee, S. (2008). Building a service-oriented ontology for wireless sensor networks. In *Proc. 7th IEEE/ACIS International Conference on Computer and Information Science ICIS2008*, pages 649–654. IEEE.
- Kin-Pong Chan, F., Wai-chee Fu, A., and Yu, C. (2003). Haar wavelets for efficient similarity search of time-series: With and without time warping. *IEEE Transactions on Knowledge and Data Engineering*, 15(3):686–705.
- Komazec, S., Cerri, D., and Fensel, D. (2012). Sparkwave: continuous schema-enhanced pattern matching over RDF data streams. In *Proc. 6th ACM International Conference on Distributed Event-Based Systems DEBS 2012*, pages 58–68. ACM.
- Korn, F., Jagadish, H. V., and Faloutsos, C. (1997). Efficiently supporting ad hoc queries in large datasets of time sequences. In *Proc. ACM SIGMOD international conference on Management of data*, pages 289–300. ACM.
- Koubarakis, M. and Kyzirakos, K. (2010). Modeling and querying metadata in the semantic sensor web: The model stRDF and the query language stSPARQL. In *Proc. 7th Extended Semantic Web Conference ESWC 2010*, pages 425–439. Springer.
- Lane, N., Miluzzo, E., Lu, H., Peebles, D., Choudhury, T., and Campbell, A. (2010). A survey of mobile phone sensing. *Communications Magazine, IEEE*, 48(9):140–150.
- Laney, D. (2001). 3D data management: Controlling data volume, velocity, and variety. Technical report, META Group.
- Le-Phuoc, D., Dao-Tran, M., Xavier Parreira, J., and Hauswirth, M. (2011a). A native and adaptive approach for unified processing of linked streams and linked data. In *Proc. 10th International Semantic Web Conference ISWC 2011*, pages 370–388. Springer.

## References

---

- Le-Phuoc, D., Parreira, J., Hausenblas, M., Han, Y., and Hauswirth, M. (2010). Live linked open sensor database. In *Proc. 6th International Conference on Semantic Systems I-Semantics 2010*, pages 1–4. ACM.
- Le-Phuoc, D., Quoc, H., Parreira, J., and Hauswirth, M. (2011b). The linked sensor middleware—connecting the real world and the semantic web. Technical report, Semantic Web Challenge 2011.
- Legeay, N., Roantree, M., Jones, G., O'Connor, N., and Smeaton, A. (2007). Semi-automatic semantic enrichment of raw sensor data. In *Proc. OTM 2007 Workshops: On the move to meaningful internet systems*, pages 13–14. Springer-Verlag.
- Lenzerini, M. (2002). Data integration: a theoretical perspective. In *Proc. 21st ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems PODS 2002*, pages 233–246. ACM.
- Lerner, A. and Shasha, D. (2003). Aquery: Query language for ordered data, optimization techniques, and experiments. In *Proc. 29th international conference on Very large data bases VLDB 2003*, pages 345–356. VLDB Endowment.
- Lesh, N., Zaki, M., and Ogihara, M. (1999). Mining features for sequence classification. In *Proc. 5th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 342–346. ACM.
- Levy, A. (1998). The information manifold approach to data integration. *IEEE Intelligent Systems*, 13:12–16.
- Lewis, M., Cameron, D., Xie, S., and Arpinar, B. (2006). ES3N: A semantic approach to data management in sensor networks. In *Proc. 1st International Workshop on Semantic Sensor Networks SSN 2006 at ISWC*.
- Lin, J., Keogh, E. J., Wei, L., and Lonardi, S. (2007). Experiencing SAX: a novel symbolic representation of time series. *Data Mining and Knowledge Discovery*, 15(2):107–144.
- Liu, L., Pu, C., and Tang, W. (1999). Continual queries for internet scale event-driven information delivery. *IEEE Transactions on Knowledge and Data Engineering*, 11(4):610–628.
- Lubyte, L. and Tessaris, S. (2009). Supporting the development of data wrapping ontologies. In *Proc. 4th Asian Conference on The Semantic Web ASWC'09*, pages 31–45. Springer-Verlag.
- Luckham, D. and Vera, J. (1995). An event-based architecture definition language. *IEEE Transactions on Software Engineering*, 21(9):717–734.



- Madden, S., Shah, M., Hellerstein, J., and Raman, V. (2002). Continuously adaptive continuous queries over streams. In *Proc. ACM SIGMOD international conference on Management of data*, pages 49–60. ACM.
- Madden, S. R., Franklin, M. J., Hellerstein, J. M., and Hong, W. (2005). TinyDB: an acquisitional query processing system for sensor networks. *ACM Transactions on Database Systems*, 30(1):122–173.
- Mansouri-Samani, M. and Sloman, M. (1997). GEM: A generalized event monitoring language for distributed systems. *Distributed Systems Engineering*, 4:96–108.
- Mei, Y. and Madden, S. (2009). Zstream: a cost-based query processor for adaptively detecting composite events. In *Proc. ACM SIGMOD International Conference on Management of data*, pages 193–206. ACM.
- Möller-Levet, C., Klawonn, F., Cho, K., and Wolkenhauer, O. (2003). Fuzzy clustering of short time-series and unevenly distributed sampling points. *Advances in Intelligent Data Analysis V*, 2810:330–340.
- Moraru, A. and Mladenice, D. (2012). A framework for semantic enrichment of sensor data. In *Proc. 34th International Conference on Information Technology Interfaces ITI 2012*, pages 155–160. IEEE.
- Osuna, E., Freund, R., and Girosi, F. (1997). An improved training algorithm for support vector machines. In *Proc. IEEE Workshop Neural Networks for Signal Processing NNSP 1997*, pages 276–285. IEEE.
- Papaioannou, T. G., Riahi, M., and Aberer, K. (2011). Towards online multi-model approximation of time series. In *Proc. 12th International Conference on Mobile Data Management MDM 2011*, pages 33–38, Washington, DC, USA. IEEE.
- Patni, H., Henson, C., and Sheth, A. (2010). Linked sensor data. In *Proc. 2010 International Symposium on Collaborative Technologies and Systems CTS 2010*, pages 362–370. IEEE.
- Paton, N. and Díaz, O. (1999). Active database systems. *ACM Computing Surveys*, 31(1):63–103.
- Paulino, H. and Santos, J. (2011). A middleware framework for the web integration of sensor networks. *Sensor Systems and Software*, 57:75–90.
- Pérez, J., Arenas, M., and Gutierrez, C. (2009). Semantics and complexity of SPARQL. *ACM Transactions on Database Systems*, 34(3):1–45.
- Pérez de Laborda, C. and Conrad, S. (2006). Database to semantic web mapping using RDF query languages. In *Proc. 25th international Conference on Conceptual Modeling*, pages 241–254. Springer-Verlag.

## References

---

- Pérez-Urbina, H., Horrocks, I., and Motik, B. (2009). Efficient query answering for owl 2. In *Proc. 8th International Semantic Web Conference ISWC 2009*, pages 489–504. Springer-Verlag.
- Poggi, A., Lembo, D., Calvanese, D., Giacomo, G. D., Lenzerini, M., and Rosati, R. (2008). Linking data to ontologies. *Journal on Data Semantics*, 10:133–173.
- Prud’hommeaux, E. (2007). SPASQL: SPARQL Support In MySQL. <http://www.w3.org/2005/05/22-SPARQL-MySQL/XTech>.
- Prud’hommeaux, E. and Seaborne, A. (2008). SPARQL query language for RDF, W3C recommendation. Technical report, World Wide Web Consortium. <http://www.w3.org/TR/rdf-sparql-query/>.
- Ramakrishnan, R. and Silberschatz, A. (1998). Scalable integration of data collections on the web. Technical report, University of Wisconsin-Madison Department of Computer Sciences.
- Rinne, M., Törmä, S., and Nuutila, E. (2012). Sparql-based applications for rdf-encoded sensor data. In *Proc. 5th International Workshop on Semantic Sensor Networks SSN2012 at ISWC*, volume 904, pages 81–96. CEUR-WS.org.
- Rodríguez, A., McGrath, R., Liu, Y., Myers, J., and Urbana-Champaign, I. (2009). Semantic management of streaming data. In *Proc. 2nd International Workshop on Semantic Sensor Networks SSN 2009 at ISWC*, pages 80–95. CEUR-WS.org.
- Rosati, R. (2012). Prexto: query rewriting under extensional constraints in DL-lite. In *Proceedings of the 9th international conference on The Semantic Web*, pages 360–374. Springer-Verlag.
- Rosati, R. and Almatelli, A. (2010). Improving query answering over dl-lite ontologies.
- Ruckhaus, E., Calbimonte, J., García-Castro, R., and Corcho, O. (2012). Short paper: From streaming data to linked data—a case study with bike sharing systems. In *Proc. 5th International Workshop on Semantic Sensor Networks SSN2012 at ISWC*, volume 904, pages 109–114. CEUR-WS.org.
- Rundensteiner, E. A., Ding, L., Sutherland, T. M., Zhu, Y., Pielech, B., and Mehta, N. (2004). Cape: Continuous query engine with heterogeneous-grained adaptivity. In *Proc. 30th international Conference on Very Large Data Bases VLDB 2004*, pages 1353–1356. VLDB Endowment.
- Russomanno, D., Kothari, C., and Thomas, O. (2005). Sensor ontologies: from shallow to deep models. In *Proc. 37th Southeastern Symposium on System Theory*, pages 107–112. IEE.

- Sahoo, S. S., Halb, W., Hellmann, S., Idehen, K., Jr, T. T., Auer, S., Sequeda, J., and Ezzat, A. (2009). A survey of current approaches for mapping of relational databases to RDF. Technical report, W3C RDB2RDF Incubator Group. [http://www.w3.org/2005/Incubator/rdb2rdf/RDB2RDF\\_SurveyReport.pdf](http://www.w3.org/2005/Incubator/rdb2rdf/RDB2RDF_SurveyReport.pdf).
- Sathe, S., Papaioannou, T., Jeung, H., and Aberer, K. (2012). Efficient sensor data management techniques. deliverable d1.3. Technical report, PlanetData FP7.
- Schimak, G. and Havlik, D. (2009). Sensors anywhere - sensor web enablement in risk management applications. *ERCIM News*, 76:40–41.
- Schmidt, M., Hornung, T., Lausen, G., and Pinkel, C. (2009). Sp<sup>2</sup>bench: a sparql performance benchmark. In *Data Engineering, 2009. ICDE'09. IEEE 25th International Conference on*, pages 222–233. IEEE.
- Schreier, U., Pirahesh, H., Agrawal, R., and Mohan, C. (1991). Alert: An architecture for transforming a passive dbms into an active dbms. In *Proc. 17th International Conference on Very Large Data Bases VLDB'91*, pages 469–478. VLDB Endowment.
- Seaborne, A., Steer, D., and Williams, S. (2007). SQL-RDF. <http://www.w3.org/2007/03/RdfRDB/papers/seaborne.html>.
- Sequeda, J. F., Arenas, M., and Miranker, D. P. (2012). On directly mapping relational databases to RDF and OWL. In *Proc. 21st international conference on World wide web WWW 2012*, pages 649–658, New York, NY, USA. ACM.
- Shneidman, J., Pietzuch, P., Ledlie, J., Roussopoulos, M., Seltzer, M., and Welsh, M. (2004). Hourglass: An infrastructure for connecting sensor networks and applications. Technical Report TR-21-04, Harvard University.
- Stonebraker, M., Çetintemel, U., and Zdonik, S. B. (2005). The 8 requirements of real-time stream processing. *ACM SIGMOD Record*, 34(4):42–47.
- Sullivan, M. and Heybey, A. (1998). Tribeca: a system for managing large databases of network traffic. In *Proc. USENIX Annual Technical Conference ATEC'98*, pages 2–2, Berkeley, CA, USA. USENIX Association.
- Tappolet, J. and Bernstein, A. (2009). Applied temporal RDF: Efficient temporal querying of RDF data with SPARQL. In *Proc. 6th Extended Semantic Web Conference ESWC 2009*, pages 308–322. Springer-Verlag.

## References

---

- Taylor, K. and Leiding, L. (2011). Ontology-driven complex event processing in heterogeneous sensor networks. In *Proc. 8th Extended Semantic Web Conference ESWC 2011*, pages 285–299. Springer-Verlag.
- Terry, D., Goldberg, D., Nichols, D., and Oki, B. (1992). Continuous queries over append-only databases. In *SIGMOD '92*, pages 321–330. ACM.
- Wei, W. and Barnaghi, P. (2009). Semantic annotation and reasoning for sensor data. In *Proc. 4th European Conference on Smart Sensing and Context EuroSSC 2009*, pages 66–76. Springer-Verlag.
- Wiederhold, G. and Genesereth, M. (1997). The conceptual basis for mediation services. *IEEE Expert: Intelligent Systems and Their Applications*, 12(5):38–47.
- Witt, K., Stanley, J., Smithbauer, D., Mandl, D., Ly, V., Underbrink, A., and Metheny, M. (2008). Enabling sensor webs by utilizing SWAMO for autonomous operations. In *Proc. 8th Eighth Annual NASA Earth Science Technology Conference ESTC 2008*.
- Xi, X., Keogh, E., Shelton, C., Wei, L., and Ratanamahatana, C. (2006). Fast time series classification using numerosity reduction. In *Proc. 23rd International conference on Machine learning ICML'06*, volume 150, pages 1033–1040. ACM Press.
- Xing, Z., Pei, J., and Keogh, E. J. (2010). A brief survey on sequence classification. *ACM SIGKDD Explorations Newsletter*, 12(1):40–48.
- Yao, Y. and Gehrke, J. (2002). The Cougar approach to in-network query processing in sensor networks. *ACM SIGMOD Record*, 31(3):9–18.
- Ye, L. and Keogh, E. (2009). Time series shapelets: a new primitive for data mining. In *Proc. 15th ACM SIGKDD international conference on Knowledge discovery and data mining KDD'09*, pages 947–956. ACM.
- Zhang, Y., Duc, P., Corcho, O., and Calbimonte, J.-P. (2012a). SRBench: A Streaming RDF/SPARQL Benchmark. In *Proc. 11th International Semantic Web Conference ISWC 2012*, pages 641–657. Springer.
- Zhang, Y., Duc, P. M., Groffen, F., Liarou, E., Boncz, P., Kersten, M., Calbimonte, J.-P., and Corcho, O. (2012b). Benchmarking RDF storage engines. Deliverable D1.2. Technical report, PlanetData FP7.