UNIVERSITY OF SOUTHAMPTON

FACULTY OF PHYSICAL SCIENCES AND ENGINEERING

School of Electronics and Computer Science

Incremental Rule-based Reasoning on Semantic Data Streams

by

Rehab Albeladi

Thesis for the degree of Doctor of Philosophy

July 2016

UNIVERSITY OF SOUTHAMPTON

ABSTRACT

FACULTY OF PHYSICAL SCIENCES AND ENGINEERING

School of Electronics and Computer Science

Thesis for the degree of Doctor of Philosophy

INCREMENTAL RULE-BASED REASONING FOR SEMANTIC DATA STREAMS

Rehab Albeladi

This thesis investigates the area of semantic stream processing, in which data streams are combined with semantic reasoning techniques. We have investigated techniques for rule-based reasoning over semantic streams in which reasoning is implemented natively over streams as data flow networks, and have developed an adaptive optimisation method to cope with the changing nature of streams. The contributions of this thesis include R4, a native rule-based reasoner for RDF streams using the Rete algorithm, and a cost-based adaptive plan optimiser designed for RDF streams. We have evaluated the performance of R4 and compared it to both a typical static reasoner and to the state-of-the-art in stream reasoners. The results show that R4 significantly outperforms these reasoners in terms of throughput. We have also evaluated the adaptive optimisation technique, with results that show the ability of the optimiser to devise and adopt better performing plans at runtime.

Table of Contents

List of Tables

List of Figures

List of Listings

DECLARATION OF AUTHORSHIP

I, Rehab Albeladi, declare that this thesis and the work presented in it are my own and have been generated by me as the result of my own original research.

Incremental Rule-based Reasoning on Semantic Data Streams

I confirm that:

1. This work was done wholly or mainly while in candidature for a research degree at this University;
2. Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated;
3. Where I have consulted the published work of others, this is always clearly attributed;
4. Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work;
5. I have acknowledged all main sources of help;
6. Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself;
7. Parts of this work have been published as: [please list references below]:

Signed: .....

Date: .....

Acknowledgements

I would like to thank my family for their endless love and support, in particular my mother Saleha and father Awadh, who do everything they can to support me and have encouraged me through my journey. Special thanks also to my beloved husband Abdulrahman for his continuous support and for maintaining a balance between his responsibility as a father to our children and his own PhD project (I congratulate him for passing his viva last week!). I am also thankful to my son Yazeed and my miracle daughter Serene who have had to excuse me when I have been busy and have not had enough time for them.

In the academic sphere, I would like to express my sincerest gratitude to my supervisor Dr Nicholas Gibbins who has supported me throughout my doctorate with critical thinking and guidance in every step. I was lucky to have been supervised by him. Many thanks also for my second supervisor, Prof. Kirk Martinez for the help and support, especially with the streaming dataset from the SemsorGrid4Env project.

I also wish to thank the Government of Saudi Arabia and Taibah University for their Scholarship programme, which provided me with a generous scholarship and continuing financial support.

Abbreviations

IoT	Internet of Things
DBMS	Data Base Management System
DSMS	Data Stream Management System
CEP	Complex Event Processing
RDF	Resource Description Framework
RDFS	RDF Schema
OWL	Web Ontology Language
SPARQL	SPARQL Protocol And RDF Query Language
RIF	Rule Interchange Format
IRI	Internationalised Resource Identifier
RSP	RDF Stream Processing
SSN	Semantic Sensor Network
CCO	Channel Coast Observatory

Chapter 1: Introduction

New developments in the realm of the Internet and Web are constantly evolving. On the one hand, the growing use of sensors and embedded devices has given rise to a vision of a future Internet called The Internet of Things (IoT) which aims to interconnect all these devices through a global network (Atzori et al., 2010). By providing Internet connectivity to "smart" embedded devices, computers will be able to automatically identify, monitor, react to, and perform actions with regard to everyday objects. The IoT will use technology such as radio frequency identification (RFID) (Finkenzeller, 2010) to uniquely identify objects and apply different event processing mechanisms to process the data streams generated by devices.

According to Sunddamaeker et al., (2010), the number of Internet-connected personal computers (PCs) was approximately 1.5 billion, in addition to over 1.0 billion Internet-connected mobile phones. When the present "Internet of PCs" moves towards the IoT, 50–100 billion devices will be connected to the Internet by 2020 (Sundamaeker et al., 2010). The IoT has received considerable interest, and has been extensively investigated by academia (Atzori et al., 2010), industry (Da Xu et al., 2014), and governments (Vermesan et al., 2011) in order to achieve its objectives.

The proposed middleware architecture for the IoT in recent years has often followed the service-oriented architecture (SOA) approach (Atzori et al., 2010). SOA principles allow complex systems to be decomposed into applications consisting of simpler and more well-defined components. Using common interfaces and standard protocols, SOA helps with the integration of enterprise applications. However, SOA solutions are often too heavy for devices with limited capabilities (Guinard et al., 2009).

Another suggested approach is to integrate real-world objects into the World Wide Web using resource-oriented architecture to build the Web of Things. Guinard et al. (2010) applied the representational state transfer (REST) architectural style to create loosely coupled services on the Web so that they can be easily reused. According to this approach, smart objects are becoming first-class citizens on the Web as every object is identified by a Uniform Resource Identifier (URI).

On the other hand, as the current Web is becoming the largest medium of information, many researchers are working on the Semantic Web, a vision of the future Web which

aims to enable computers to understand the meaning of Web content (Berners-Lee et al., 2001). This will enable software agents to access the Web and carry out intelligent tasks on behalf of the user. Data in the Semantic Web have to be given well-defined meanings to be machine processable. A number of formats have been standardised, such as Resource Description Framework (RDF) (Cyganiak et al., 2014), RDF Schema (Brickley et al., 2014), and Web Ontology Language (OWL) (Hitzler et al., 2012). The aim of these formats is to structure and give semantics to the Web data, which will then enable automatic reasoning and processing of this data.

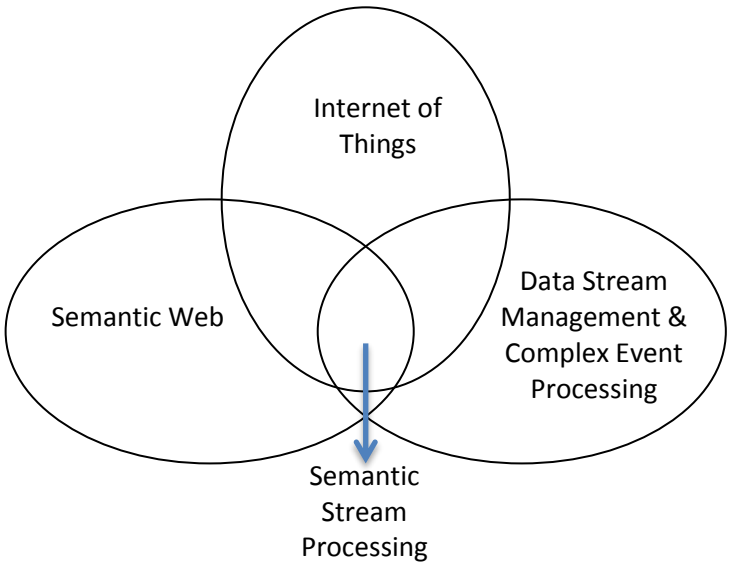
Despite the fact that the IoT focuses on infrastructure issues and the Semantic Web places more emphasis on knowledge representation, as they basically work in different layers, the two visions aim to interlink the virtual and physical worlds. The IoT commonly identifies real-world objects using unique RFID tags, while the Semantic Web uses URIs to uniquely identify real-world objects. However, both visions could complement each other.

Data in such areas are continuously and sometimes massively produced by applications that are becoming more and more data driven. For instance, the increasingly popular sensing devices are currently used to generate environmental observations, monitor patient conditions, track locations, and observe energy consumption (Fang et al., 2014; Myung et al., 2002; Wei and Li, 2011). On the Web side, microblogging services, such as twitter, also deliver real-time streaming data (Sakaki et al., 2010). This class of data differs significantly from the typical, mostly static, data model, with changes being the rule rather than the exception. This static data model (typically a relational model) does not consider the temporal and ordered nature of data streams. Data in this model are stored and *ad hoc* queries are issued to process the data in a pull-based fashion. Streaming data, on the contrary, can arrive at a higher rate than that at which they can be stored and processed, and therefore need to be handled on the fly in a push-based manner, using continuous queries. A special class of management systems, called data stream management systems (DSMSs), has appeared to provide such functionality (Babcock et al., 2002).

While DSMS engines can be used to process data streams generated by the IoT devices, the heterogeneity of the data sources and formats makes interoperability and integration a real challenge. As the Semantic Web's standard formats (RDF and OWL) and linked data principles (Bizer et al., 2009) facilitate the data integration of heterogeneous datasets, there have been efforts to lift stream data to a semantic level (Sheth et al., 2008). Incorporating semantics would also enable inferencing capabilities, a feature that is not available in DSMSs owing to the lack of semantics in their data stream models.

Inference enables the automatic discovery of new information (Lucas and Van Der Gaag, 1991). Data are modelled as a set of relationships between resources, so the inferencing process on the Semantic Web means the generation of new relationships based on the dataset and some background knowledge in the form of vocabularies or rule sets. A number of reasoning engines for the Semantic Web, with different expressivity levels, have appeared, such as Bossam (Jang and Sohn, 2004), Pellet (Sirin et al., 2007), and FaCT++ (Tsarkov and Horrocks, 2006). However, they mostly assume the knowledge base to be static or slowly changing. Enabling reasoning upon rapidly changing semantic streams would require a similar shift as that from database management systems to data stream management systems. An example where inference is needed to derive new facts in a streaming setting is described in (Cugola and Margara, 2012) as follows: in a smart building, there could be a rule that produces an event of fire when it observes a very high temperature happening in a short time window with a smoke event. This rule needs to be continuously applied using a time-aware model to changing streams of events.

The processing of semantically-annotated dynamic data gives rise to semantic stream processing, which has received increased interest from the Semantic Web community over the past few years[1]. We consider Semantic Stream Processing to lie at the intersection of the Data Stream Management research, and the Semantic Web area because it builds its techniques on both of them. The Internet of Things area is also relevant, as it can serve as a context in which applications generate and process streaming data (Figure 1.1).



[1] A W3C community group for RDF stream processing (RSP) was created in 2013. Available at: https://www.w3.org/community/rsp/

Figure 1.1: Background research areas

Semantic stream processing aims to provide the abstractions, foundations, methods, and tools required to integrate data streams and semantic processing systems (Della Valle et al., 2009). Challenges in this area include, but are not limited to, the following:

- Defining a data model for semantic streams that reflects their ordered and temporal nature
- Supporting continuous queries
- Integrating background data and dynamic streams
- Providing reasoning capabilities
- Dealing with incomplete and noisy streams
- Distributed stream processing.

While most of these challenges are similar to those faced by the database community when developing stream processing systems, these efforts cannot be directly applied to the semantic stream processing area because of the primary differences between the relational models used in the DSMSs and the RDF graph-based model of semantic data. A number of semantic stream processing systems (Barbieri et al., 2010a; Bolles et al., 2008; Calbimonte et al., 2010; Anicic et al., 2011; Le-Phuoc et al., 2011) have been developed in this area and mostly address the continuous queries challenge through the extension of SPARQL (Harris et al., 2013), the Semantic Web's query language. They also defined an extension of the RDF data model to express the temporal element.

In this research, we mainly focused on the reasoning aspect, with the aim of enabling rule-based reasoning over RDF data streams as continuously running data flow networks. We propose the use of an incremental reasoning framework with low-level operators directly applied over the RDF streams. This RDF-native approach offers maximum optimisation opportunities, which have a major impact on response time, memory consumption, and completeness of results. We also investigated the optimisation problem itself, i.e. translating a set of rules into a set of processing data flow networks. We consider the adaptivity of such network structures to be an essential requirement to cope with the constantly changing nature of the data streams.

That said, we addressed the rule-based reasoning on semantic streams with a number of limitations. Firstly, streams in real-world applications are potentially delayed, incomplete, imprecise, or noisy. This can lead to query results of unknown quality, so methods to clean

the data are required, as are algorithms when reasoning with uncertainty. To limit the scope of this research, we assumed that the arriving streams were ordered and complete. Secondly, continuous query languages in the literature usually enable the addition and deletion of stream elements, which then requires truth maintenance methods to ensure that the data remain consistent. We assumed that the input streams were of the append-only type and that the generated results were stream instances too.

1.1 Research Hypotheses

The main focus of this research was on enabling efficient and effective reasoning over semantic streams. Efficient processing refers to the way in which the system handles the input data. Data streams usually arrive in high volumes and velocity, so input throughput is an important measure of stream processing system performance. The resources within a stream processing system must be managed carefully owing to the high volumetric nature of streams and because storing all the incoming data is usually impractical. On the other hand, effectiveness measures are reflected by the quality of results produced by the system. Precision and recall (the correctness and completeness of the retrieved results) are commonly used metrics in general information retrieval systems. The timeliness of the results is another effectiveness metric of prime importance in streaming applications, especially as the results lose their value in the event of long delays. Stream processing systems should be highly responsive.

We formulated our objectives based on the following research question and hypotheses:

How could we efficiently (by using minimal resources and ensuring high throughput) and effectively (by providing timely results with high precision and recall) enable rule-based reasoning over RDF data streams using data flow networks?

- *Hypothesis 1:* It was anticipated that our continuous reasoning approach would improve throughput and responsiveness, when compared to a traditional static reasoner.
- *Hypothesis 2:* It was expected that the trade-off between the completeness of the output results and the processing time could be controlled by varying the resource allocation.
- *Hypothesis 3:* It was anticipated that resource usage would be reduced by our approach of enabling node sharing, where possible, between the rules.

- *Hypothesis 4:* It was anticipated that system performance would be improved by monitoring the characteristics of the streams in order to re-organise the reasoning networks.

1.2 Contributions

The contributions of this thesis are:

- A native rule-based reasoner for RDF streams using dataflow networks. Our reasoner supports RIF Core (Boley et al., 2013), which corresponds to the language of definite Horn rules without function symbols, equivalent to Datalog. While CQELS (Le-Phuoc et al., 2011) and Sparkwave (Komazec et al., 2012) (state-of-the-art RDF stream processing systems) support native processing of RDF streams, CQELS does not support reasoning, and Sparkwave provides a lower expressivity of a subset of RDF Schema.
- A cost-based adaptive optimiser designed for RDF streams. In the RDF stream processing area, only CQELS has addressed the issue of adaptive optimisation. CQELS follows the fine-grained routing-based approach, while we follow a more coarse-grained plan-based approach.
- An extension of RIF Core that adds the window construct to the language in order to enable users to specify time constraints as part of the rules.

1.3 Thesis Structure

The remainder of this thesis is as follows: Chapter 2 provides related background material to our work, first in the area of stream processing and continuous queries, including a review of existing stream processing systems and continuous query languages (Section 2.1), and then with regard to the Semantic Web (Section 2.2), with a focus on the knowledge representation of its data and the reasoning techniques.

The literature on semantic stream processing specifically is covered in Chapter 3. There is a review of research advancement in different aspects of the field, including RDF stream processing, reasoning and inference support, publishing, distributed processing, developed environments, and benchmarking semantic stream processing engines.

Details of our reasoning framework and implemented system are given in Chapter 4. A number of general requirements to enable efficient reasoning over the data streams are

provided in Section 4.1. The adopted reasoning framework is then presented in Section 4.2, starting with a justification of the design decisions that were taken by linking them to the requirements. An RDF stream data model is then defined, followed by an operational description of different operators needed for the continuous processing of the data streams. R4, our rule-based reasoner for RDF streams, using the Rete algorithm (Forgy, 1982), is covered in Section 4.3. The rule language supported by the system is discussed first, followed by the system architecture and finally several aspects of data processing within the system. The different nodes through which the data passes through from input to output are detailed, together with other related issues, such as data structures and garbage collection.

An evaluation of the implemented system is then presented in Chapter 5. An evaluation scenario, including a description of the input data sets, together with a number of designed use cases to test system functionality are described first in Section 5.1. Then, Section 5.2 details a number of experiments that have been carried out in order to comparatively evaluate the system's performance, first against a static reasoner (to test the first hypothesis), and then against state-of-the-art semantic stream processing systems that support inference.

Chapter 6 then addresses the optimisation problem. We first describe the initial plan generation using a static optimiser in Section 6.1, and then present our adaptive optimisation approach in Section 6.2. We present the cost model that was used to estimate the expense of implementing the plans and two optimisation algorithms that were utilised to generate more cost-effective plans. Finally, we describe the monitoring and plan migration employed in the system.

An evaluation of the optimisation process is presented in Chapter 7. Section 7.1 tests the effects of reducing window sizes on the quality of results. The operator sharing optimisation technique is evaluated in Section 7.2. Section 7.3 is dedicated to evaluate the adaptive optimiser. It first verifies the cost model by comparing the actual and estimated costs of different plans. Then, the costs of the adaptive plans generated by the adaptive optimiser are compared to those of the static plans, in stable and unstable conditions.

Finally, concluding remarks are provided in Chapter 8, and the contributions of this research, together with suggested future work, are highlighted.

1.4 Publications

Parts of this research have been published, as follows:

- Albeladi, R., Martinez, K. and Gibbins, N. (2015) Incremental rule-based reasoning over RDF streams. In *RDF Stream Processing Workshop at the 12th Extended Semantic Web Conference, Portoroz, Slovenia*.
- Albeladi, R. (2012) Distributed reasoning on semantic data streams. In *International Semantic Web Conference* (pp. 433–436). Springer Berlin Heidelberg.
- Albeladi, R., Martinez, K. and Gibbins, N. (2012) Distributed stream reasoning. At *the poster track of the 9th Extended Semantic Web Conference, Heraklion, Greece.*

Chapter 2: Background Research

The Semantic Stream Processing area builds on the knowledge in both the Data Stream Processing and the Semantic Web areas. In this chapter, we present a background review of both of these areas. Section 2.1 reviews the research in the Stream Processing area, highlighting the differences between a database management system and a data stream processing system, briefly introducing some of the developed stream processers, and discussing some techniques, including continuous queries, adaptive optimisation, and distributed processing of streaming data. The Semantic Web is then reviewed in Section 2.2, with a focus on semantic data formats and related reasoning techniques.

2.1 Data Stream Processing

The emergence of data-intensive applications such as sensor networks and IoT applications has created a number of requirements that cannot be easily met with the traditional database systems, due to the sheer amount of data with which they operate (Stonebraker et al., 2005). These applications require the ability to capture, analyse, and react to events on a real-time basis. Data in these applications needs to be modelled as transient data streams, rather than as typical persistent relations.

A data stream is a real-time, continuous sequence of items, ordered implicitly by arrival time or explicitly by timestamps (Golab and Ozsu, 2003). Data streams differ from stored relations in several ways: the data elements in the stream arrive online, they are potentially unbounded in size, the system has no control over the arrival order of the data elements, and the data elements may be discarded after being processed. These differences raise a number of challenges which have been addressed by many researches in the database community (Arasu et al., 2003; Abadi et al., 2003; Hammad et al., 2004).

2.1.1 Data Stream Management Systems

DBMS vs. DSMS: In order to describe a data stream management system (DSMS), we compare it to a conventional database management system (DBMS). DBMSs are being successfully used in a wide variety of domains and applications. They use the ‘store and query’ model, where they permanently store the data and then evaluate queries against the stored data (Garcia-Molina et al., 2000). Data stream applications do not fit this model, as the stream elements can arrive faster than they can be stored (Golab and Ozsu, 2003).

Therefore, a new specialised class of management systems that can perform real-time analysis over streaming data has appeared. In Data Stream Management Systems (DSMS), data arrives as one or more continuous data streams where a special type of queries –called continuous queries—can be performed on them. Continuous queries (Terry et al., 1992) are similar to traditional database queries, except they are issued only once and then run continually over the changing data. When newly arriving data matches the running continuous query, new results are returned to the client. Figure 2.1 depicts the general model of data stream management systems.

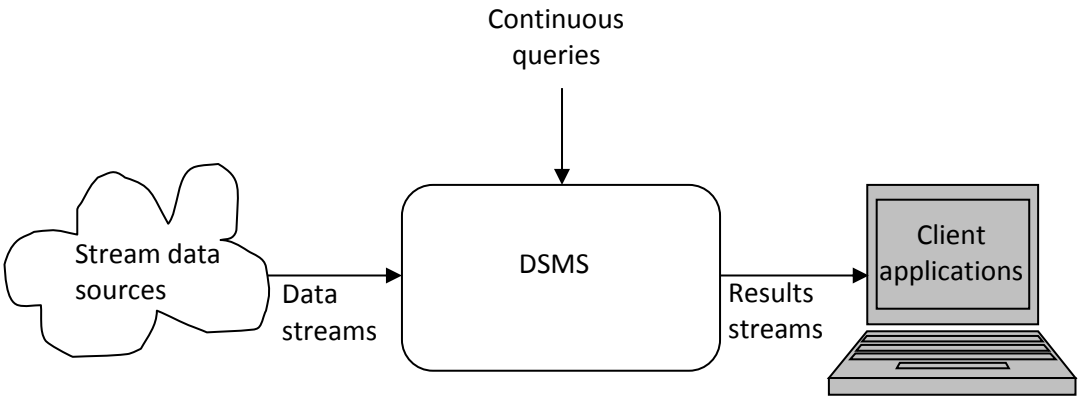


Figure 2.1: DSMS model

The operational model of a DSMS differs from that of a DBMS. In DBMSs, humans actively update the dataset and initiate queries, and the DBMS acts as a passive repository, simply executing the queries on the dataset. Abadi et al. (2003) refer to this as a human-active, DSMS-passive (HADP) model. In contrast, DSMSs receive data from various external sources, such as sensors, and not from humans. The DSMS plays an active role, continuously processing this data and alerting humans when anomalies are detected. Therefore, this is called the DBMS-active, human-passive (DAHP) model (Abadi et al., 2003).

DBMS and DSMS are not only different in their basic models, but also in the nature of the data that they handle and the queries that can be performed on this data. The following table (2.1) illustrates the major differences between them (Abadi et al., 2003, and Babcock et al., 2002).

Table 2.1: Comparison between DBMS and DSMS

	DBMS	DSMS
Model	HADP (Human Active DBMS Passive)	DAHP (DBMS Active Human Passive)
	Only the current state of the data is important	The history of data is also important
	Applications require no real-time services	Applications have real-time requirements
	Triggers and alerters are second class citizens	Trigger-oriented applications
Data	Persistent relations	Transient streams
	Random access	Sequential access
	Relatively low update rate	Possibly high arrival rate
	Data elements are synchronised	Data arrives asynchronously
Queries	One-time queries	Continuous queries
	Queries have exact answers	Answers computed with incomplete information

Existing DSMS: There are a large number of applied data stream management systems. Some of them are briefly introduced below, while their continuous query languages are described later.

TelegraphCQ (Chandrasekaran et al., 2003) is an adaptive query engine for sensors to process queries effectively. TelegraphCQ is focused on meeting the challenges of handling large numbers of continuous queries over high-volume, highly variable data streams. TelegraphCQ differs from other data stream systems due to its focus on extreme adaptivity and the novel infrastructure required to support such adaptivity.

Aurora (Abadi et al., 2003) is a data stream management system for monitoring applications. It is basically a data-flow system where data elements flow through a loop-free, directed graph of processing operators. Aurora performs compile-time and run-time optimisation. Moreover, it detects resource overhead, and it performs load-shedding.

STREAM (Arasu et al., 2003) is a general-purpose data stream management system produced by Stanford University. It translates declarative queries into physical query plans composed of operators, queues, and synopses. STREAM also performs load-shedding, if needed, by introducing approximations. Furthermore, it provides a graphical interface to enable users to monitor and manipulate query plans as they run.

Nile (Hammad et al., 2004) is the query engine of the "STEM" stream database system. Nile performs efficient pipelined execution of sliding window queries over multiple streams. Another feature of Nile is its scalability in terms of the sizes of both queries and data streams. It also provides guarantees for Quality of Service and Answers. Further features include support for approximation and integrated online data mining tools.

After this first generation of stream processing engines was developed in academia, a number of commercial and open-source large-scale streaming data analytics platforms appeared, including Apache Storm[2], Yahoo's S4[3], and Spark Streaming[4]. Storm was the first distributed stream processing system to gain traction throughout research and practice (Wingerath et al., 2016). It was developed by Twitter in late 2010 and eventually became an Apache project in 2014. Storm's architecture is based on the master-workers paradigm. Optimised for low latency, Storm excels at high speed and is able to perform in the realm of single-digit milliseconds in certain applications (Wingerath et al., 2016).

In contrast to previous stream processing systems, which work on a tuple at a time, Spark Streaming (Zaharia et al., 2013) uses a micro-batching approach. As an extension of Spark[5], which is a fast and general-purpose cluster computing system, Spark Streaming chunks incoming streams into small batches, forming Spark's Resilient Distributed Datasets (RDDs). An RDD is an immutable, deterministically recomputable, distributed and fault-tolerant dataset (Zaharia et al., 2012). In contrast to Storm, Spark Streaming is optimised for high throughput (Kipf and Kemper, 2016).

PipelineDB[6] is an open-source project that continuously runs SQL queries on streaming data. It extends the database system PostgreSQL[7] by introducing the concept of continuous views. A continuous view differs from a regular view as it selects inputs from a combination of streams and tables and is incrementally updated in real time as new data is written to those inputs. PipelineDB excels at SQL queries that reduce the cardinality of streaming datasets through summarisation and aggregation.

A recent development in the area is the ReStream project (Schleier-Smith et al., 2016), which was developed at the University of California, Berkeley. ReStream is designed for

[2] http://storm.apache.org/
[3] http://incubator.apache.org/s4/
[4] http://spark.apache.org/streaming/
[5] https://spark.apache.org/
[6] https://www.pipelinedb.com/
[7] https://www.postgresql.org/

accelerated replay. It processes stored event logs in parallel, with throughput much higher than the real-time rate. This enables rapid development of applications that require backtesting, as developers can evaluate new functionalities using weeks, or months worth of stored data in minutes. ReStream combines streaming semantics with the performance characteristics of batch processing, enabling serial equivalent processing of stored logs using distributed computing resources.

2.1.2 Querying Data Streams

Queries over data streams have much in common with queries in DBMSs. However, there is an important difference in the query execution model between one-time queries and continuous queries. One-time queries as used in traditional database systems are evaluated once over a point-in-time snapshot of the data set, with the answer returned to the user after the query evaluation has finished. In contrast, the class of queries used with data streams are called continuous queries. These queries are issued once, operate continuously over a period of time and incrementally return new results over time as new data arrives (Babu and Widom, 2001). Continuous queries' different nature raises a number of challenges, such as their unbounded memory requirements, the need for approximate query answering, the problem of blocking operators, and the need to reference past data (Babcock et al., 2002). Some relevant challenges are described below.

As the data streams can be of unbounded size, query answers potentially require unbounded memory to store them. Moreover, even if there is no need to store the answers, as they can be provided as data streams themselves, some queries need unbounded memory to compute exact results, e.g. join operators. Although there are some external memory algorithms (Vitter, 1999) that can handle data that require a larger memory space than the main memory, these algorithms are too slow for getting real-time responses. Arasu et al., (2003) differentiated between queries that can be answered exactly given a bounded memory, and those which need unbounded memory. In the latter case, providing approximate answers is a possible solution, which is in itself another challenge.

There are several approaches to approximate query answers. These include using sliding windows, batch processing, sampling and synopses. Instead of evaluating the query over the whole history of data, the first approach evaluates queries on windows of recent data. According to Babcock et al., (2002), this method has several attractive properties. It is a well-defined, easily understood, deterministic method. More importantly, it emphasises recent data, which is usually more important to users, over old data in most real-world

applications. For example, to make sense of network traffic patterns or sensor data, insights based on recent data will be more useful than insights based on old data. In this sense, windows can be thought of not as an approximation technique reluctantly applied due to the inability to process queries over all historical data, but rather as part of the desired query semantics explicitly expressed by the user.

As in Arasu et al., (2003), windows can be tuple-based, where an integer parameter is set to be the number of returned tuples with the largest time-stamp. Alternatively, these can be time-based windows, where a time interval parameter is set and the returned tuples should have time-stamps falling in this interval range. Windows can also be classified based on the movements of their endpoints: two fixed endpoints define a fixed window, two sliding endpoints define a sliding window, while one fixed and one moving endpoint define a landmark window (Golab and Ozsu, 2003).

To slide the window over its elements, there are two methods. The first is to specify a periodic value, either a number of tuples or time units, depending on the window type, at which the window updates its content and causes a new evaluation. This value is usually called a slide or step, e.g. a sliding window over the last ten minutes with a step of two minutes. The second method is the eager or data-driven approach, in which the window is updated automatically whenever a new element arrives into the stream, which causes another evaluation of the query. This means that it will generate results as soon as a new tuple arrives; in the first approach, there is an added delay equal to the step value.

In terms of performing continuous queries over sliding windows, there are also two main approaches: query re-evaluation and incremental evaluation (Ghanem et al., 2007). In the first approach, the query is re-evaluated over a snapshot of the stream, representing the current window independently from previous windows each time the window is updated. This approach, while it has clear semantics and is easy to implement, can result in redundant processing of certain tuples because they can be parts of several consecutive windows. Conversely, in the incremental evaluation approach, only the changes in the window relative to the previous window are processed by the query operators.

Applying windows on streams might not always be enough to handle the unbounded memory requirement, for two reasons. First, window sizes can be large enough that the entire contents of the window cannot be buffered in memory. Second, the stream arrival rate can be faster than the time needed by the system to both update the window and process the newly arrived element (Babcock et al., 2002). Therefore, other approximation

techniques are needed. In the former case, it is impractical to try to answer a query over all the data. Instead, queries can be evaluated over a sample of the data stream, producing approximate answers that may be sufficient for some applications (e.g. (Demaine et al., 2002)). In the latter case, eager (or streaming) evaluation of queries, in which windows are updated and queries are re-evaluated upon arrival of each new stream element, might not be appropriate. The natural solution is to process the incoming data in batches (Babcock et al., 2002) as in the XJoin algorithm (Urhan and Franklin, 2000). Rather than producing a continually up-to-date answer, the data elements are buffered as they arrive, and the answer to the query is computed periodically as time permits. Batch processing is a good approach when streams are bursty; a system that cannot keep up with the peak stream rate may be able to handle the average rate, while buffering the streams at peak time and catching up during a slow period. In contrast to sampling, this approach does not cause any uncertainty about the accuracy of the answer, but does sacrifice timeliness instead.

A closely related challenge is the blocking operator's problem. A blocking query operator is an operator that is unable to produce its first output tuple until it has seen its entire input (Babcock et al., 2002). Sort, aggregates, and some implementations of the join operator are considered blocking. For example, the Nested Loop Join (NLJ) needs to scan the entire inner relation and compare each tuple therein with the current tuple of the outer relation. Since data streams maybe infinite, a blocking operator that takes input from a data stream will never see its entire input, and will never be able to produce any output. Blocking operators can be unblocked using the same windowing technique described above, restricting the streaming input to a finite window. To avoid re-scanning the entire window, these operators need to support incremental evaluation. Several unblocked join algorithms have appeared that can process inputs in an incremental, pipelined approach, such as the Symmetric Hash Join (SHJ) (Wilschut and Apers, 1993), and XJoin (Urhan and Franklin, 2000). The basic scheme of these algorithms is that they build hash tables on the fly for each of their inputs, and when a tuple arrives from one of them, it is inserted into the corresponding table and the other tables are probed for matches.

Stream query languages: A number of stream query languages have been proposed. They fall into three different paradigms: relation-based languages, object-based languages, and procedural languages (Golab and Ozsu, 2003).

Relation-based languages typically have SQL-like syntax and enhanced support for windows and ordering. CQL (Arasu et al., 2006), StreaQuel (Chandrasekaran, 2002), and AQuery (Lerrner and Shasha, 2003) are three relation-based languages. CQL (Continuous

Query Language) has been used in the STREAM project (Arasu et al., 2003). It considers streams as time-stamp ordered relations. It defines three types of operators: stream-to-relation, relation-to-relation, and relation-to-stream operators. Stream-to-relation operators produce a relation from a stream using different types of windows, relation-to-relation operators are standard relational algebra operators producing a relation from another relation, while relation-to-stream operators produce a stream from a relation. CQL introduces three relation-to-stream operators: Istream, to indicate an insertion to the output stream, Dstream, to indicate a deletion, and Rstream that returns all tuples in the relation at a certain time. In contrast, StreaQuel – used in TelegraphCQ (Chandrasekaran et al., 2003) – does not have relation-to-stream operators, since it considers all inputs and outputs as streams.

Object-based languages also have SQL-like syntax, but add support to streaming abstract data types (ADTs). This approach is used in the COUGAR system (Bonnet et al., 2001) where each type of sensor is modelled as an ADT with special signal-processing methods. Another approach is used by the Tribeca system (Sullivan and Heybey, 1998), where stream contents are classified according to a type hierarchy.

Table 2.2: Stream query languages features (Golab and Ozsu, 2003)

Language/ system	Motivating applications	Allowed inputs	Basic operators	Supported windows			Custom operators?
				Type	Base	Execution	
AQuery	Stock quotes, network traffic analysis	Stored relations	Relational, "each", order-dependant (first, next, etc.)	Fixed, landmark, sliding	Time and count	Not discussed	Via "each" operator
Aurora	Sensor data	Streams only	σ, π, ∪, ⋈, group-by, resample, drop, map, window sort	Fixed, landmark, sliding	Time and count	Streaming	Via map operator
CQL/ STREAM	All-purpose	Streams and relations	Relational, relation-to-stream, sample	Sliding	Time and count	Streaming	Allowed
StreaQuel/ TelegraphCQ	Sensor data	Streams and relations	Relational	All types	Time and count	Streaming or periodic	Allowed
Tribeca	Network traffic analysis	Single input stream	σ, π, group-by, union aggregates	Fixed, landmark, sliding	Time and count	Streaming	Allows custom aggregates

In a procedural query language – as an alternative to a declarative query language – the flow of the data can be specified by users. Aurora DSMS (Abadi et al., 2003) has a user interface where users can create query plans by joining a set of boxes (operators) by arcs, to specify the desired data flow. Aurora distinguishes between order-agnostic operators, including filter, map, union and order-sensitive operators, such as BSort, aggregate, join, and resample.

Table 2.2, from Golab and Ozsu (2003), summarises the features of stream query languages. In general, relation-based languages appear to be more popular.

2.1.3 Continuous Query Optimisation

In a relational DBMS, a query processor consists of a number of components (Garcia-Molina et al., 2000). First, a parser is used to parse SQL queries into an internal representation, such as query graphs. Then a query optimiser is used to produce a query plan, usually in a form of a tree, which specifies how exactly the query is to be evaluated. The nodes in the tree represent relational algebra operators (e.g. select, join, sort, etc.) and the arcs represent data being generated and consumed by these operators. The plan is then fed into an execution engine to be executed at run time.

The query processor performance relies heavily on the optimiser. A query can usually be evaluated using different equivalent query plans that give the same answer but may differ significantly in their response time and consumed memory (Garcia-Molina et al., 2000). Thus, the optimiser's task of choosing the most efficient plan is crucial. To find the optimal plan, optimisers use statistics about the tables involved in the query (e.g. cardinalities, histograms), estimate operators' selectivities, and employ a cost model to estimate the total cost of different plans.

This approach doesn't work well in Data Stream Management Systems for two main reasons. First, the statistics about streaming data are usually unknown at compile time, or even impossible to obtain in case of unbounded streams, therefore, traditional cardinality-based cost metrics are not applicable (Kang et al., 2003). Second, for long-running queries, data characteristics such as stream input rates might change during the query execution, which will cause the optimal plan to change. Hence, the optimise-then-execute approach does not suit this likely changing environment (Babu and Bizarro, 2005). For streaming, federated or any inherently uncertain environment, research has since introduced adaptive

query processers, in which runtime feedback is used to adapt query processing (Deshpande et al., 2007).

2.1.3.1 Cost models

Because traditional cost metrics do not apply to continuous queries, Kang et al. (2003) introduced a unit-time basis cost model to estimate the performance of different sliding window join algorithms. The model considers different tasks performed by a window join (i.e. inserting, probing, invalidating) to find the cost of handling an individual input tuple of each input stream separately. This cost is then multiplied by the left (right) stream input rate to obtain the per-unit-time cost of the left (right) part of the join. The left and right costs are simply added together to calculate the join's total cost. This division between the left and right parts means that the algorithms used to perform them are completely independent, i.e. the left join can be a hash join while the right part uses a nested loop algorithm.

Ayad and Naughton (2004) extended this unit-time-based join's cost to model the cost of conjunctive queries with sliding windows with output rates and window size estimations of all operators. A similar model was used in Cammert et al. (2008) to adaptively manage resources by adjusting window sizes and time granularities.

Another model introduced in Viglas and Naughton (2002) also considered that inputs to continuous queries are streams with input rates, as opposed to relations with known cardinalities in traditional queries, to shift from cardinality-based models to a rate-based model. They presented formulas to estimate output rates of different operators in a query plan, which can be used to either optimise for the highest output rate or to identify which plan will produce a specified number of results in the shortest amount of time.

More recent work on cost models has concentrated on distributed stream processing. For example, Zeitler and Risch (2011) present a cost model for a stream splitting operator that splits streams into parallel sub streams, while the model presented in (Heinze et al., 2014) can estimate latency caused by operator movements across multiple sites. However, our work is based on a centralised setting, in which the previously described models represent the state of the art.

2.1.3.2 Adaptive optimisation

Adaptive query optimisation techniques vary significantly in many factors. They differ in what they attempt to adapt to (e.g. memory fluctuations, data arrival rates, user preferences), the aim of adaptivity (e.g. minimising response time, maximising throughput), the nature of feedback they collect, the frequency of feedback collection and plan altering (inter-query, intra-query, per tuple), and the effect of adaptivity (i.e. what behaviour it can change) (Gounaris et al., 2002). We summarise the main adaptive approaches, organised by increasing frequency of adaptivity, first in the traditional store-then-query context, then in continuous streaming systems.

In early optimisers, such as the System R query optimiser (Selinger et al., 1979), the statistic-gathering scheme was very coarse-grained, running only periodically – typically once a day or once a week – requiring administrative commands, and consuming significant resources (Hellerstein et al., 2000). To enhance this scheme, Adaptive Selectivity Estimation (ASE) was proposed (Chen and Roussopoulos, 1994), in which statistics are gathered on a per-query level. For each query, the system tracks the sizes of intermediate results and uses them to refine statistical metadata for future optimisation decisions. While still moderately coarse, this inter-query method enables the system to learn more and perform better in subsequent queries.

On an intra-query level, blocking operators offer materialisation points at which actual statistics of intermediate results can be obtained, and the rest of the plan can be adapted. Optimisation and execution are interleaved by dividing a query plan into stages, executing one plan stage to completion, and using the statistics gathered from this stage to optimise the execution of the next one (Deshpande et al., 2007). Mid-query re-optimisation (Kabra and DeWitt, 1998) also operates on an intra-query level, employing progressive optimisation and proactive re-optimisation. Instead of plan staging, it initially optimises the entire plan, inserts statistics-gathering operators at specified checkpoints (usually after blocking operators), and dynamically re-optimises the downstream (i.e. remaining) parts of the plan if the runtime statistics differ significantly from the original estimates. A similar approach is the Corrective Query Processing (CQP) (Ives et al., 2004) used in the Tukwila integration system. However, CQP relies on pipelined operators – such as the XJoin operator (Urhan and Franklin, 2000) – instead of blocking operators, so the execution cost is continuously monitored.

On the same level of frequency, but specifically designed for continuous stream processors, Babu et al., (2004) proposed an adaptive ordering algorithm for pipelined commutative stream filters, called A-Greedy (for Adaptive Greedy). A-Greedy adds two logical components to the query engine: a profiler, and a re-optimiser. The profiler continuously collects and maintains statistics about the recent input tuples. These statistics are used by the re-optimiser to detect and correct suboptimal correlated filters ordering. The algorithm can be also applied to a multiway stream join (MJoin) (Viglas et al., 2003), which is a generalisation of symmetric binary joins. However, fully pipelined MJoins do not hold internal states, which causes the performance to suffer due to the excessive re-computation of intermediate results. To tackle this problem, an A-Caching algorithm was introduced (Babu et al., 2005), which places subresult caches adaptively in MJoins to minimise re-computation. A-Greedy and A-Caching are implemented in StreaMon (Babu and Widom, 2004), the adaptive query processing engine for the STREAM DSMS.

The Telegraph stream project (Chandrasekaran et al., 2003) introduced a more revolutionary method to enable adaptive processing on the very fine-grained level of per-tuple planning, i.e. each tuple could be processed using a different plan. The basic idea behind this approach is to treat query execution as a process of routing tuples through pipelined operators, and to adapt by changing the order in which tuples are routed through, effectively resulting in changing plans at a tuple level. They introduced the eddy dataflow operator (Avnur and Hellerstein, 2000), which encapsulates adaptivity. Data flows into the eddy from input streams or relations, and the eddy routes tuples into pipelining operators, which return tuples back to the eddy after processing. The eddy sends the tuple to the output only when all the operators have handled it. It also monitors the execution continuously, to adaptively choose the routing order for each tuple. The eddy operator can implement different routing policies; the initial one proposed by Avnur and Hellerstein is the lottery scheduling routing policy (2000). In this policy, the eddy operator monitors the input and output queues of each operator, assigning and penalising tickets to the operators for every sent and returned tuple. The eddy then holds a lottery among the eligible operators, routing the tuple to the operator that has the biggest number of tickets.

While increasing the level of adaptivity's frequency can achieve better optimal plans, it can also increase the overhead in terms of the time spent in the optimisation process and the amount of memory needed to store all the required statistics. This trade-off needs to be carefully considered in designing an adaptive optimiser. In other words, the potential benefits of adaptivity should be weighed against the additional overhead they incur.

2.1.4 Distributed Stream Processing

The previously presented data stream management systems perform centralised processing of streams, as the early efforts in this domain have focused on designing new operators and languages. Researchers then considered extending these systems to support distributed processing of data streams, which are usually physically distributed, in order to achieve a better scalability and higher availability (Shah et al., 2004). Researchers have built on and extended the work of parallel query processing (Yu et al., 1993) in order to handle some challenges that are not usually present in traditional database systems, mainly imposed by the dynamic nature of streaming data.

Distributed query processing can take two forms. First, the query plan operators can be distributed among several machines, so that each machine executes a different sub-tree of the complex query tree. This method is called inter-operator parallelism, vertical parallelism, or simply pipelining. The other method is called intra-operator parallelism, horizontal parallelism, or partitioning, in which data is partitioned across multiple machines rather than the query. In this approach, the same operator is copied on the participating machines, and each of these instances operates on part of the data.

To distribute a query processing over multiple machines, stateless operators (e.g. filter, project) can be relatively easily pipelined. Therefore, early work in distributed query processing in the traditional store-then-query model has focused on parallelising individual, traditional, state-full operators such as the hybrid-hash join and sort. However, in these efforts, the distribution mechanism needs to be included in every operator implementation. Graefe (1990) then introduced a novel operator to abstract the distribution mechanism called the exchange operator. This operator encapsulates all parallelism issues and therefore makes implementation of parallel database algorithms significantly easier and more robust. The exchange operator is inserted between the producer and consumer operators to ensure proper routing of data and can provide vertical or horizontal parallelism. However, the exchange operator uses static partitioning techniques, such as hash partitioning, range partitioning, or round robin, which means that they do not adapt to load variation at runtime.

Inspired by the exchange operator, but for the continuous processing model, two intra-operator adaptive partitioning operators were introduced at U.C. Berkeley: RiverDQ (Arpaci-Dusseau et al., 1999) and Flux (Shah et al., 2003). The distributed queue abstraction RiverDQ addresses the load-balancing problems for a limited set of operators:

those for which the partitioning of the input stream can be content-insensitive, such that any tuple can be sent to any instance of the consuming operator. Every input tuple is routed to a randomly chosen consumer instance weighted by the emptiness of the queue to that instance. As this approach cannot be applied to content-sensitive operators such as joins and group-by aggregates, the Flux operator generalises both Exchange and RiverDQ to encapsulate the logic of online partitioning for a wide range of content-sensitive operators. Following the design of the Exchange operator, each Flux operator is composed of two parts: Flux-Cons (Flux consumer) and Flux-Prod (Flux producer). Flux-Cons is essentially an iterator, while Flux-Prod encapsulates the routing logic. Flux follows a centralised approach in which a central controller decides when to move which load where. Unlike Exchange, these decisions are made online adaptively, based on real-time collected statistics. A Flux operator can handle short-term imbalances using a buffering and reordering mechanism, and adapts to long-term imbalances by enabling online repartitioning and transferring of accumulated states.

In terms of distributed DSMSs, Borealis (Abadi et al., 2005) is a second-generation distributed stream processing engine. In contrast to Flux, Borealis enables inter-level parallelism, and works on a fully distributed, peer-to-peer network. Borealis derives its core stream processing functionality from Aurora (Abadi et al., 2003), and distribution functionality from its preceding project, Medusa (Cetintemel, 2003). However, Borealis extends both of them by adding more features. On the processing side, it supports dynamic revision of query results, and dynamic modification of queries. On the distribution side, Borealis provides a scalable, Quality of Service (QoS)-based resource allocation and optimisation, as well as fault tolerance. Borealis addresses these issues in the domain of sensor networks, which adds the challenges of simultaneously optimising different QoS metrics, such as processing latency, throughput, or sensor lifetime, and the ability to perform optimisations at different levels of granularity: a node, a sensor network, a cluster of sensors and servers, etc.

The above-described approaches provide the essential parallelism architecture and functionality. However, they have to employ a load-balancing algorithm. Typically, a dynamic load-distributing algorithm has four components (Shivaratri et al., 1992): a transfer policy, a selection policy, a location policy, and an information policy. A transfer policy determines whether a node is qualified to participate in a load transfer, either as a sender or as a receiver. Once a node is determined to be a sender, a selection policy then chooses which task or data partition to transfer. A location policy is then responsible for

finding a good receiver for the selected task. In adaptive load-balancing algorithms, this decision should be made at runtime, based on statistics about current load on other nodes. The type of statistics about the states of the system's operators, when they should be collected, and where they are to be collected from, should all be specified in the information policy.

The policy for load balancing in Flux proceeds in rounds. Each round consists of two phases: a statistics collection phase, and a move phase. At the beginning of each round, the central controller asks all Flux-Cons instances to start collecting statistics, and specifies a duration parameter which indicates the time that should be spent collecting information before returning it to the controller. The information to be collected is the amount of time spent idle during this phase, and the number of tuples processed per partition. This information is used by the controller to make a list of pair-wisely associated operators. The most loaded server is paired with the least loaded server, and so on. The controller then performs some threshold tests to ensure that moving a partition from the first operator of a pair to the second operator would not increase the imbalance between them. After that, the second phase starts by halting the producer operator. Meantime, the state is transferred from the first Flux-Cons instance to the second. As soon as this completes, the producer operator is resumed and sends data to the new receiver instance.

In the Borealis system, each site contains a local monitor, a local optimiser, and a neighbourhood optimiser, which together are responsible for continuously optimising the allocation of query network fragments to processing sites. Local monitors maintain various operator and site-level statistics regarding utilisation and queuing delays for various resources, including CPU, disk, bandwidth and power. These statistics are periodically forwarded to end-point monitors that run at every site that produces final outputs, and are responsible for evaluating QoS for every output message. There is also a global optimiser that reacts to alerts from an end-point monitor indicating a problem which an output QoS measures, including lifetime, throughput, and latency problems. On a local level, when a monitor detects specific resource bottlenecks, the corresponding optimisers either request the node to shed loads (ordered by the local optimiser) or, preferably, identify slack resources to offload to other sites (determined by the neighbourhood optimiser). The tasks chosen to move are those that improve resource utilisation most while imposing the minimum load migration overhead. Borealis uses a correlation-based load distribution algorithm (Xing et al., 2005) to minimise average load variation and maximise average load correlation, which will accordingly result in small average end-to-end latency.

2.1.5 Complex Event Processing

As a result of the broad-spectrum of application domains that require on-the-fly processing of information, several research communities have addressed this problem, with each bringing its own expertise, point of view and vocabulary (Cugola and Margara, 2012). The data stream processing model discussed in the previous subsections was developed by the database research community and can be seen as an evolution of the traditional data processing supported by DBMS. DSMSs resemble DBMSs in processing incoming data through a sequence of transformations using common SQL operators based on relational algebra such as selections, aggregates and joins. In contrast, other research communities, including those of distributed information systems, business process automation, control systems and network monitoring, contribute to the complex event processing model (CEP), which, according to Cugola and Margara (2012), can be routed to the publish-subscribe domain. In complex event processing, data in input streams are viewed as simple events that can be filtered, combined and transformed into composite events of interest to users. Complex event queries can check for occurrence and non-occurrence of composite events by imposing temporal, logical and value-based constraints over streaming events.

An example of a complex event processing system is SASE (Wu et al., 2006), which is a monitoring system of streams of RFID readings encoded as events. SASE employs a declarative complex event language that enables filtering and correlating events to match specific patterns. The structure of the language consists of three clauses: the EVENT clause which specifies the event pattern, i.e. the events that need to be detected and the sequential and logical relations between them; the WHERE clause, which defines value-based constraints; and the WITHIN clause, which enables specificity of window sizes. The SEQ operator in this system is an example of a temporal constraint that is at the heart of complex event processing and is not usually supported by the stream processing model, which primarily focuses on producing and continuously updating query answers.

Another example is Esper[8], which is considered to be the leading open-source CEP provider (Cugola and Margara, 2012). Esper is distributed as embeddable components written in Java and C#, which makes it suitable for integration into any Java- or .NET-based process. It defines a rule language called the Event Processing Language (EPL), which enables joining, filtering, sorting, aggregating, grouping, merging, and splitting

[8] http://www.espertech.com/esper/

event series or streams, as well as detecting sequences and patterns. In addition, it offers a rich set of temporal windows that can be parameterised, including sliding, tumbling, partitioned and named windows. Esper's engine processing model is based on dynamic state machines and delta networks, in which only changes to data are communicated across object boundaries.

2.2 The Semantic Web

Over the past years, vast amounts of information have been published on the Web, making it a universal source of information. However, as current web pages are designed to be read by humans, computers are not capable of analysing this information. The Web has developed rapidly as a medium of documents for people rather than data that can be processed automatically. Given the current extent of information on the Web – answers for numerous questions are out there – it is true that search engines do a very complex task of finding and ranking related web pages, based on keyword matching, but it needs a human to find the exact answer (Antoniou and Van Harmelen, 2004).

Transferring the available information on the Web into machine processable formats is the aim of the Semantic Web (Berners-Lee et al., 2001). The Semantic Web is defined by Tim Berners-Lee et al. (2001, p. 29) as "an extension of the current Web, in which information is given well-defined meaning, better enabling computers and people to work in cooperation". Enabling computer programs to understand the meanings of web content will enable the software agents to access the Web and carry out intelligent tasks on behalf of the user. Mechanisms for shared understanding enable machines – or software agents – from different domains to communicate with each other, automating large parts of users' lives.

Giving meanings – or semantics – to the available information includes structuring and annotating data, and adding logic and inference capabilities. These features had been extensively studied long before the Web was developed, in the area of Artificial Intelligence (Brachman and Levesque, 1985). However, the traditional knowledge representation systems have usually been small: limited to questions that can be answered reliably, and centralised: requiring everyone to share exactly the same definitions (Berners-Lee et al., 2001). On the other hand, the Semantic Web should be as big and decentralised as the current Web, which leads to another important aim of the Semantic Web: knowledge interoperability and easy information integration, taking advantage of the Web's unique

naming and universality (Berners-Lee et al., 2001). A number of languages have already been standardised to achieve these goals.

The Semantic Web layer cake, shown in Figure 2.2, represents a number of important technologies that have been developed in order to actualise the Semantic Web vision. These include RDF (Cyganiak et al., 2014) to express data, RDFS (Brickley et al., 2014) and OWL (Hitzler et al., 2012) to enable shared understanding of concepts, SPARQL (Harris et al., 2013) to query the data, and RIF (Kifer and Boley, 2013) to add interchangeable rules support to the Semantic Web.

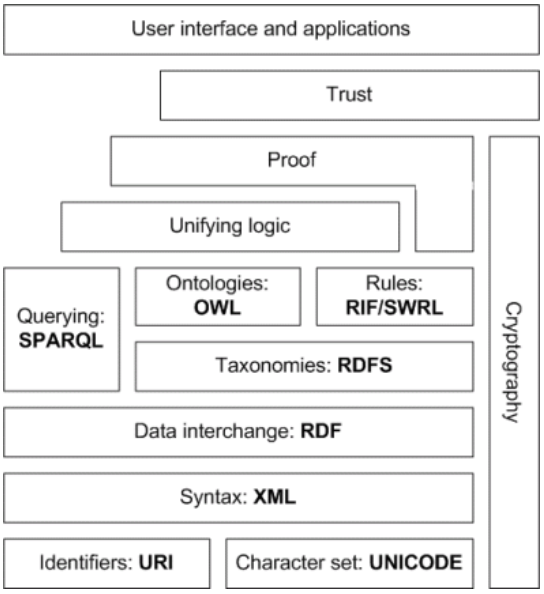


Figure 2.2: The Semantic Web layer cake (Domingue et al., 2011)

As we aim to enable reasoning on semantic data streams, we first provide a broader look at knowledge representation in general, and their related reasoning methods, with a focus on the rule-based approach. Then we describe the above-mentioned Semantic Web standards and technologies. Finally, a number of existing reasoners for Semantic Web data are reviewed.

2.2.1 Knowledge Representation and Reasoning Techniques

Reasoning about knowledge in Artificial Intelligence is used to discover new facts from a knowledge base (Lucas and Van Der Gaag, 1991). Different inferencing algorithms have been developed for different Knowledge Representation formalisms. These formalisms include logic (Metakides and Nerode, 1996), production rules (Newell, 1973), semantics networks (Woods, 1975), and frames (Minsky, 1975). They have different features and so can serve different systems' requirements in terms of expressivity and performance.

Logical representation of knowledge includes two basic forms: propositional logic and predicate logic. While propositional logic is limited in representing real-world knowledge, predicate logic has the ability to use variables and functions. Due to its high expressivity, the predicate logic is undecidable, meaning that it is not guaranteed that the proof procedure will terminate (Lucas and Van Der Gaag, 1991). Description Logics (Baader, 2003) were then produced as a decidable subset of predicate logic with lower expressivity. Description logic provides the following reasoning services: satisfaction, subsumption, and classification (Donini et al., 1996).

Knowledge can also be represented in the form of production rules that are widely used in expert systems, which are knowledge-based systems that can offer solutions to specific problems in a given domain at a level comparable to that of an expert in the field (Lucas and Van Der Gaag, 1991). Rules are easy to understand, maintain, and to derive inference from. Each rule consists of two parts: a precondition (IF part), and an action (THEN part). In rule-based engines, the condition parts of rules are checked with the current state of the world (working memory). If a match occurs, the action part of the matched rule is executed.

This inferencing process can be done through two mechanisms: forward chaining, and backward chaining. Forward chaining (Forgy, 1981) is a data-driven algorithm as it starts with data and looks for rules which apply to the facts until a goal is reached. While this approach can result in a large number of entailments that will never be queried, queries can get fast responses as all the entailments are asserted at the insertion time. On the other hand, backward chaining is a goal-driven algorithm (Shortliffe, 1976), as it starts with a goal and looks for rules that apply to that goal until a conclusion is reached. Backward chaining can often be very expensive to support interactive-time query satisfaction; therefore, forward chaining is more efficient in dynamic situations that require real-time responses (Buchanan and Duda, 1983).

Checking conditions of rules with working memory is the core process of any rule engine. A naive implementation would be to check each coming fact against each rule, which usually results in a very slow system when dealing with large numbers of rules or facts (Forgy, 1982). The Rete algorithm was developed by Forgy (1982) to provide a basis for a more efficient implementation. It is a dataflow network-based algorithm designed to speed the pattern matching process.

The Rete algorithm can process large data sets efficiently because it avoids iterating over both data elements (facts or working memory) and over the production rules. To avoid iteration over data elements, the Rete algorithm stores with each condition (or pattern), a list of the data elements that it matches. These lists are updated when the working memory changes, in a forward chaining process. To avoid iteration over the production rules, the Rete algorithm uses a tree-structured network to represent the rules. The network is composed of different types of operators, which are also called nodes. The main two types can be called the "intra-element" operators such as the filter operator and the "inter-element" operators such as the join operator. An important feature of the Rete algorithm is that when two patterns require the same nodes, these nodes are shared rather than building duplicate ones. The tree-like network divides the matching process into multiple steps that perform different checks, so if a data element does not match the first node, it is simply discarded and does not complete its way through the network.

To illustrate with an example, consider the following pattern of a working memory element, written in OPS5 language:

```
(Expression: ^Name <N>, ^Arg1 0, ^Op +, ^Arg2 <X>)
```

This arithmetic expression has four attribute-value pairs, with attributes indicated by the symbol ^ and values representing either constants or variables. In OPS5, variables are written between brackets, like <N> in the example. The intra-element features checked by this pattern are: the element class must be Expression, the Arg1 value must be zero, and the Op value must be +. On the other hand, the value of Name is an inter-element feature, as it needs to be matched with variables from other patterns. For example, the Plus0x rule from (Forgy, 1982), which aims to simplify algebraic expressions that add zero to a number, can be written as follows:

```
(Plus0x (Goal: ^Type Simplify, ^Object <N>)
        (Expression: ^Name <N>, ^Arg1 0, ^Op +, ^Arg2 <X>)
         => (modify 2 ^Arg1 NIL ^Op NIL))
```

The rule name is followed by two patterns, representing the left-hand side, and then the symbol =>, which is in turn followed by the action that represents the right-hand side. The latter action assigns NIL to the operation and first argument values of the working memory element that matches the second pattern in the rule, leaving only the second argument as a result. In this case, the value of the Object attribute of the goal must be equal to the value of the Name attribute of the expression. Figure 2.3 shows a Rete network for the Plus0x rule (Forgy, 1982); the blue boxes represent the intra-element operators while the brown one represents the inter-element operator.

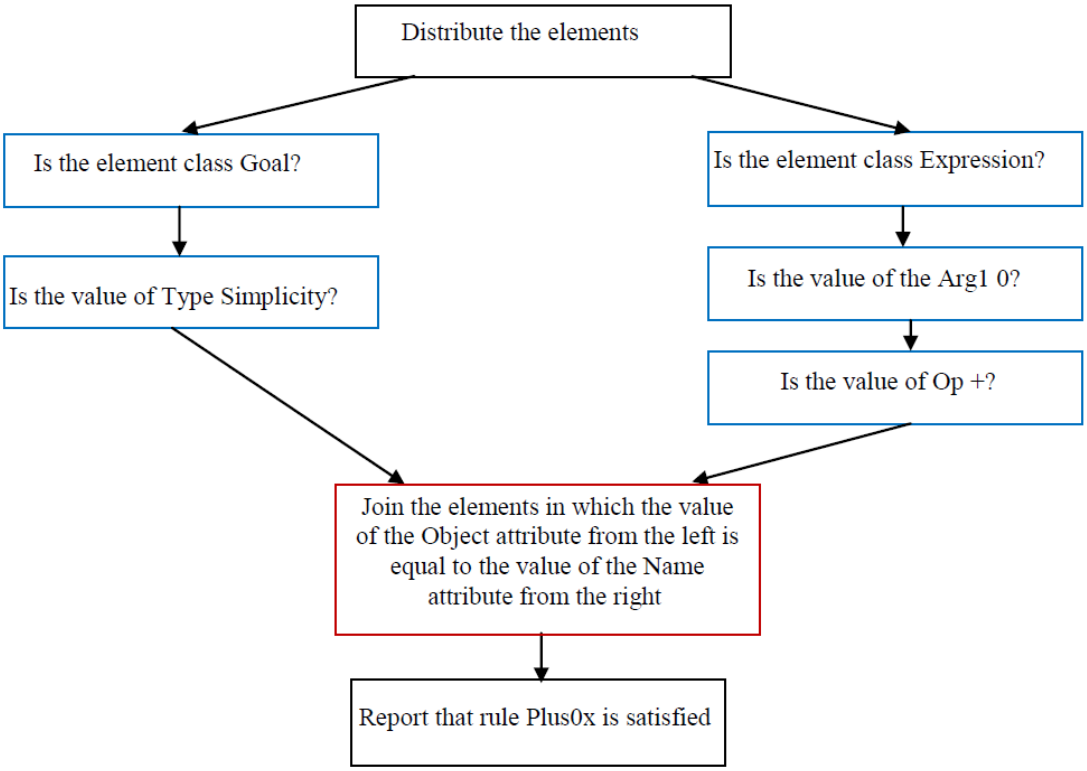


Figure 2.3: Rete network example

While different applications can add more node types, the basic ones are the root node, one-input nodes (intra-element, or alpha nodes), two-input nodes (inter-element, or beta nodes), and terminal nodes. Alpha nodes resemble the select (σ) operator of relational algebra; they only propagate statements that match their condition. On the other hand, a beta node is responsible for joining some data elements of its two inputs on their shared variable, resembling the join (⋈) operator of relational algebra. This analogy to database systems is based on considering the Working Memory elements as tuples of some universal relationship in a relational database, and so the LHS of a rule in a production system is analogous to a query in a relational database language (Miranker, 1987).

One difference between database systems and production systems is that queries in traditional database systems are typically computed only once; in other words, they are single-shot queries. In contrast, rules in a production system are longer-lived and may be computed repeatedly. To minimise the re-computation time of different production cycles, production system algorithms retain state across cycles. Alpha and beta nodes in the Rete networks maintain their own lists of every matched tuple in alpha/beta memories, to provide incremental reasoning support. Therefore, Rete algorithm trades memory space for reasoning performance.

A number of other pattern matching algorithms have appeared to address this memory/speed trade-off. TREAT (Miranker, 1987), for instance, only keeps alpha memories, and its network has only one multi-join for each rule. This means that there are no saved intermediate results, and every new element added to an alpha memory needs to be joined with all other alpha memories each time. On the continuum between Rete and TREAT, there is Rete* (Wright and Marshall, 2003), an extension of Rete with TREAT as a special case. Among other features, Rete* employs a dynamic beta-memory cut mechanism. It allows an upper bound on beta memory consumption to be specified by users, so beta memories are discarded and retained at runtime depending on the current memory consumption. If a beta memory is absent but is needed to process a token, Rete* recalculates the missing memory. If the recalculation itself depends on prior joins with absent memories, Rete* also recalculates, working back up the network until if finds a stored beta memory, or it reaches the alpha memory.

The network structure in the original Rete algorithm follows a strict left-deep linear structure, in which each join node has only two inputs; where the right input is always an alpha memory and the left input a previous beta memory. TREAT has instead one join node with any number of inputs, where the join order is statically specified following the lexical order of the condition elements. As in query optimisation – see Section 2.1.3 – the structure of the network (analogous to a query plan) and the join order, affects the performance of the system, as it affects the number of partial instantiations or intermediate results. The structure of the network also affects the possible amount of join nodes sharing between different rules (Scales, 1986). Gator (Hanson and Hasan, 1993) has then appeared as a generalised discrimination network, where any rule can have any number of join nodes, each of which can have any number of inputs. Because the Gator structure is very general, an optimiser is essential to pick a good structure for a rule, depending on the environment. Gator implements a dynamic programming optimisation strategy that uses some parameters akin to DBMS optimisers, such as selectivities and cardinalities, but also takes into consideration some additional factors, such as update frequency and memory node size.

Table 2.3 summarises the characteristics of Rete, TREAT, Rete* and Gator, along with some performance remarks obtained from Nayak et al., (1988), Wright and Marshall (2003), and Hanson and Hasan (1993).

Table 2.3: Comparison of pattern matching algorithms

	Beta memories?	Network structure	Performance
Rete	Yes	Linear networks of two-input join nodes	-Faster in general -Faster for addition -Better performance for complex rules
TREAT	No	One multiple-input join node per rule	-Less memory consumption -Faster for deletion -Better performance for simple rules (6 or less condition elements)
Rete*	Yes (with dynamic cut)	Linear networks of two-input join nodes	-Maximum memory consumption defined by user -Rete*(0) is TREAT -Always faster than Rete
Gator	Depends on optimisation	No restrictions on the number of inputs or the tree structure	-Optimised networks are faster than Rete and TREAT -Near to TREAT memory consumption -Optimisation time goes up to one minute for an 11 conditions rule

As the Rete algorithm does not provide a time model, Berstel (2002) introduced an extension to the algorithm to enable temporal reasoning. Temporal reasoning involves formalisation of the notion of time in order to provide a way to represent and reason about the temporal aspects of knowledge (Vila, 1994), which is an important feature of event processing. Berstel's extension differentiates between facts – which are maintained until explicit retraction – and events – which are maintained until they expire. Each event has a timestamp, and at the presence of temporal constraints, join nodes are responsible for computing expiry dates for events propagated from a parent node, and retracting them when they are expired.

2.2.2 Knowledge Representation on the Semantic Web

Data model: In order to enable machine-understanding of the Web data, the Semantic Web uses formal knowledge representation techniques to describe the data contained on the Web. The Resource Description Framework (RDF) was originally intended as a means for processing metadata about web pages, but it has subsequently been generalised to provide a general-purpose knowledge representation framework for web data (Domingue et al., 2011). It provides interoperability between applications that exchange machine-understandable information on the Web. RDF presents a model for representing entities and relationships. This model encodes the semantics in sets of assertions called statements

made up of three parts: subject, predicate, and object, and so they are also referred to as triples. These three elements resemble the subject, verb, and object of a simple sentence (Berners-Lee et al., 2001). For example, the sentence: John works with Tom, is represented as follows:

Subject: John

Predicate: works with

Object: Tom

This triple can be diagrammed as in Figure 2.4:

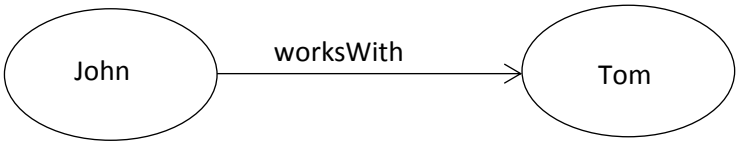


Figure 2.4: An example RDF triple

A set of statements of this form naturally forms a directed, labelled graph, in which subjects and objects can be seen as graph nodes, while predicates represent the named edges between these nodes (Cyganiak et al., 2014). An example graph is shown in Figure 2.5. Arrows represent predicates, ovals represent subjects and objects, and boxes represent a specific type of object called a literal. In contrast to XML for example, RDF graphs do not necessarily follow the tree structure, as they usually have no roots, and there is no limit to their structures.

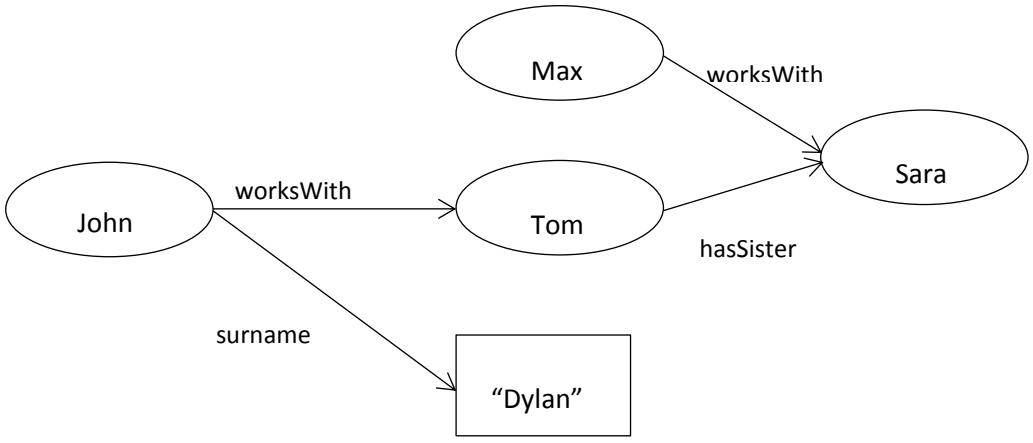


Figure 2.5: An example RDF graph

RDF triple components can be literals or resources. Literals are concrete data values such as strings or integers – for example the surname "Dylan" in the above graph – and can only appear as objects. Resources on the other hand, represent concepts and can appear as subjects, predicates, or objects. Resource names in RDF take the form of Uniform

Resource Identifiers (URIs) (Berners-Lee et al., 1998) which are unique identifiers for concepts. A special case of resource are Blank Nodes, which are implicit concepts that have no URIs or explicit names, and can only occur as subjects or objects.

RDF as a data model provides many attractive features especially for information sharing and integration over the traditional relational data model (Hebeler et al., 2009). These include the simple structure of its basic units, its unrestricted graph structure, and the global namespace provided by the use of URIs (Hebeler et al., 2009). An RDF triple with a named resource as its subject unambiguously describes that particular resource, regardless of where the triple is asserted. In contrast to the relational model, where a particular row in a database table is identified with a primary key that is unique to one table within one database, a URI is a name that is universally unique, and remains valid in any context. RDF triples are completely self-contained assertions of information, and as such they are independent from one another. This independence means that the order in which they occur is insignificant. Two RDF graphs can so be merged easily because their flexible structure does not imply any inherent significance to any one resource as compared to any other. Linking data sources together can be done simply by adding few triples to specify the relationships between the data sources, which is much simpler than the complicated schema realignment that is usually needed to integrate two data sources in a database system.

RDF is a very flexible data model, as it allows representation of any arbitrary knowledge assertions in the form of triples. There are no requirements for pre-defined data schemas as in RDBMSs, which require the definition of data structures or schemas before actual data can be asserted. This flexibility can be considered as a significant advantage when the structure of the data is not well known in advance (Taylor et al., 2006). However, this comes at a price. Pre-defined schemas offer detailed information to the DBMS on how the data is structured, which informs its decisions about data storage layouts, and query optimisation. While the RDF triple model is in itself a schema, it is still very loose compared to the detailed relational schemas. This model is analogous to a database with very few tables but with a huge number of small records.

Data extraction: RDF data are usually stored in a special type of repository called triple stores. Data from these stores can be retrieved using the W3C standardised query language, SPARQL (Harris et al., 2013), which recognises RDF as its fundamental syntax. SPARQL enables users to specify a graph pattern with variables that will be matched against a given data source, returning all matched bindings. The graph patterns themselves are composed

of independent triple patterns, which makes it possible to state, for example, that some parts of the graph are optional, or to limit any part of the entire query graph pattern to particular RDF data sets.

SPARQL for RDF data is analogous to SQL for relational databases. While the underlying graph structure is very different from the tabular format of RDMSs, a mapping between SPARQL and SQL is still possible. Cyganiak (2005) provided a transformation from SPARQL into relational algebra and outlined a translation from the relational algebra into SQL statements. For example, a triple pattern in a SPARQL query that specifies a predicate value and has subject and object as variables can be translated into a filter (σ) operation on the predicate and a projection (π) for the subject and object. Two triple patterns in a SPARQL graph pattern can be mapped into an inner join operation ($\bowtie$) on their shared value, and so on. This mapping makes existing work on query planning and optimisation available to SPARQL engines and RDF processors.

In addition, there is a substantial amount of research on native SPARQL optimisation, which draws both from the semantics of SPARQL (Pérez et al., 2006) and from approaches used in database optimisation. A SPARQL basic graph pattern optimisation using selectivity estimation is presented by Stocker et al. (2008), in which graph patterns are reordered based on a number of heuristics and summary statistics tailored for the RDF data model. The heuristics range from simple triple pattern variable counting to a more sophisticated probabilistic framework that uses pre-computed statistics to estimate the selectivities of individual and joined triple patterns. Another approach to SPARQL optimisation presented by Hartig and Heese (2007) uses syntactic rewriting based on a given SPARQL query graph model (SQGM) that supports all phases of query processing. In the query-rewriting phase, transformation rules are used to transform a SPARQL query into another semantically equivalent query in order to achieve better execution. For example, rules can be used to simplify complexly formulated queries by merging graph patterns, and can eliminate redundant or contradictory restrictions. Other approaches include providing specialised RDF indices (Harth and Decker, 2005) that can be used for selectivity computation of single triple patterns, and enabling semantic query optimisation for SPARQL (Schmidt et al., 2010).

Incorporating semantics: Another basic component of the Semantic Web are ontologies. An ontology can be defined as “a formal, explicit specification of a shared conceptualization” (Gruber, 1993). This enables computers to share not only information,

but vocabulary (Patel-Schneider et al., 2004). A typical Web ontology has a taxonomy and a set of inference rules (Berners-Lee et al., 2001). While the taxonomy defines classes of objects and relations among them, the inference rules add the power of inferring new relations. W3C has standardised RDF Schema (Brickley et al., 2014), which is a vocabulary for describing properties, classes and their hierarchies, and OWL (the Web Ontology Language) (Hitzler et al., 2012), which provides greater expressivity than RDFS.

RDF Schema is a vocabulary description language for RDF. It presents mechanisms for describing sets of related resources and the relationships between these resources (Brickley et al., 2014). So, RDFS adds some kinds of structure or schema over the unstructured RDF data. RDFS does not provide a vocabulary of application-specific classes and properties, instead, it provides the facilities needed to describe such classes and properties, and to indicate which classes and properties are expected to be used together. The main classes defined by RDFS are: Resource, Class, Literal, and Datatype. In addition, the main properties defined by RDFS are: domain, range, subClassOf, and subPropertyOf.

RDF defines an informative rule-based axiomatization of the RDFS semantics that can be executed over any RDF graph. Executing these rules will generate new facts when the existing ones match rule conditions. All the rules are stated in the form: add a triple to a graph when it contains triples matching a pattern. For example, if class X is a sub-class of Y, and class Y is sub-class of Z, then a triple stating that class X is also a sub-class of Z is asserted.

Web Ontology Language (Hitzler et al., 2012) is also a vocabulary description language; however, it is designed for applications that require greater expressivity than that provided by RDFS. OWL is a revision of the DAML+OIL ontology language (Horrocks, 2002). It adds more vocabulary to describe properties and classes, including relations between classes, cardinality, equality, richer typing of properties, and enumerated classes. The original version of OWL (Patel-Schneider et al., 2004) defines three sublanguages with increasing expressivity: OWL Lite, OWL DL, and OWL Full. While being less expressive, OWL Lite also has a lower formal complexity than OWL DL. It doesn't support some OWL features, e.g. no enumerated or disjoint classes, and puts restrictions on some other features, e.g. cardinality. OWL DL provides the maximum expressiveness that could be achieved while maintaining completeness and decidability. It supports all OWL constructs but with some restrictions. OWL Full provides the maximum expressiveness – no restrictions – but with no computational guarantees, as it can be potentially undecidable.

OWL 2 (Hitzler et al., 2012) is a revision of OWL that affords greater expressiveness through the addition of new functionalities, including keys, property chains, richer data types and data ranges. OWL 2 has three sublanguages: OWL 2 EL, OWL 2 QL, and OWL 2 RL. OWL 2 EL enables polynomial time algorithms for all the standard reasoning tasks. It can be used for applications where very large ontologies are needed, and where performance is more important than expressivity. OWL 2 QL uses standard relational database technology to enable conjunctive queries to be answered. It is particularly suitable for applications where small ontologies are used to organise large numbers of instances and where it is useful or necessary to access the data directly via relational queries. OWL 2 RL enables polynomial time reasoning using rule-extended database technologies (e.g., Datalog (Ceri et al., 1990)) operating directly on RDF triples. It can suitably serve applications where relatively small ontologies are used to organise large numbers of instances and where it is useful or necessary to operate directly on data in the form of RDF triples.

While reasoning over RDFS and OWL RL can be performed using the efficient, lightweight rule-based reasoning algorithms, other OWL dialects need Description Logic reasoning. Though OWL DL is guaranteed to be computationally decidable, that does not imply that reasoning will be completed in a realistic amount of time. The trade-off between expressivity and complexity should be taken into account when choosing the appropriate semantic representation for the application.

To map the gap between logic programs and description logic, Grosof et al. (2003) introduced new intermediate knowledge representations, Description Logic Programs (DLP) and Description Horn Logic (DHL), which fall within the expressivity intersection of rules and ontologies. They used RuleML to represent Logic programs, and OWL/DAML+OIL to represent description logics, which were the current draft standards for rules and ontologies in the Semantic Web context. They explained a bidirectional mapping of premises and inferences – called DLP-fusion – from the DLP fragment of DL to LP and from the DLP fragment of LP to DL. This fusion enables rules to be built on top of ontologies and to build ontologies on top of rules.

Adding rules: Many rule languages for the Semantic Web have appeared since its early days, to represent knowledge that either cannot be expressed in OWL or can be understood more easily with rules (Hebeler et al., 2009). These include RuleML (Boley et al., 2001), the Semantic Web Rule Language (SWRL) (Horrocks et al., 2004), in addition to rule engines with their own rule syntax such as Jena (McBride, 2002) and Jess (Friedman-Hill,

2002). As there is no one rule language that is likely to satisfy the needs of all different applications, and as one of the Semantic Web goals is to ensure interoperability between different systems, the W3C did not standardise any of them. Instead, they produced RIF: the Rule Interchange Format (Kifer and Boley, 2013) as a recommendation. RIF represents a core rule language plus extensions, which together allow rules to be translated between rule languages and thus transferred between rule systems.

To enable interchange, RIF provides multiple dialects, such as RIF Core (Boley et al., 2013), RIF Basic Logic Dialect (RIF BLD) (Boley and Kifer, 2013), or Production Rule Dialect (RIF PRD) (de Sainte Marie et al., 2013), in addition to a set of standard data types and built-in functions (RIF DTB) (Polleres et al., 2013). RIF Core is the fundamental RIF language, designed to be the common subset of most rule engines, and provides safe positive datalog with builtins. RIF BLD offers the expressive features of Horn rules, while RIF PRD is focused on the condition-response frameworks of forward-chaining rules. Although RIF dialects were designed mainly for interchange, each dialect is a standard rule language and can be used even when portability and interchange are not required.

From a theoretical viewpoint, RIF Core corresponds to the language of definite Horn rules without function symbols (often called 'Datalog') with standard first-order semantics (Boley et al., 2013). Therefore, RIF Core is a subset of RIF BLD. It is also a subset of RIF PRD, in which the conclusions of production rules are interpreted as assert actions. Syntactically, RIF Core has a number of Datalog extensions to support features such as objects and frames, similar to F-logic (Kifer et al., 1995), IRIs as identifiers of concepts, and XML Schema datatypes (Biron et al., 2004). An example of a RIF frame is <http://example.com/John> [ex:worksWith -> <http://example.com/Tom>], which corresponds to the RDF triple (<http://example.com/John>, ex:worksWith, <http://example.com/Tom>). The interoperability of RIF rules with RDF graphs and OWL ontologies is explained in the RIF, RDF and OWL Compatibility document (Bruijn and Welty, 2013), which defines the syntax and semantics of integrated RIF Core/RDF and RIF Core/OWL languages. These features make RIF Core a Web-aware language (Boley et al., 2013). Moreover, RIF Core is based on built-in functions and predicates over selected XML Schema datatypes, as specified in RIF-DTB 1.0 (Polleres et al., 2013). These include functions for comparing values, datatype-checking predicates, and basic numeric functions.

An example of a simple RIF rule document obtained from Boley et al. (2013) is presented in Listing 2.1. The rule derives 'buy' relationships from 'sell' relationships that are stored

as facts. The document can be read in English as follows: a buyer buys an item from a seller if the seller sells the item to the buyer; John sells LeRif to Mary. The conclusion that Mary buys LeRif from John can be logically derived from this statement.

```
Document(
   Prefix(cpt <http://example.com/concepts#>)
   Prefix(ppl <http://example.com/people#>)
   Prefix(bks <http://example.com/books#>)
   Group
   (
      Forall ?Buyer ?Item ?Seller (
         cpt:buy(?Buyer ?Item ?Seller) :- cpt:sell(?Seller ?Item ?Buyer)
      )
      cpt:sell(ppl:John bks:LeRif ppl:Mary)
   )
)
```

Listing 2.1: An example RIF rule document

Linked Data: While the above-described standards provide a framework for representing and processing semantic data, Linked Data principles provide a set of best practises for publishing and interlinking such data on the Web in a manner that facilitates data discovery and interoperability (Bizer et al., 2009). These principles were outlined by Berners-Lee (2006) as follows: (1) use URIs to name things, (2) use HTTP URIs to allow individuals to look up those names, (3) return useful information upon looking up URIs using standards (e.g. RDF), and (4) include links to other URIs to enable further discovery.

Following these principles, data providers add their data to a global data space that allows data to be discovered and used by various applications, creating what can be described as the web of data (Bizer et al., 2009). The Linking Open Data project[9] was established to bootstrap the web of data by converting existing open datasets to RDF and publishing them on the Web according to linked data principles. As large organisations and governments also published linked data, the number of datasets increased from 12 in 2007, at the beginning of the project, to 1,146 interlinked datasets in the most recent update of the Linked Open Data cloud (Abele et al., 2017).

2.2.3 Existing Semantic Reasoners

A semantic reasoner, rule engine, reasoning engine, or simply a reasoner, is a piece of software able to infer logical consequences from a set of facts or axioms (Singh and

[9] https://www.w3.org/wiki/SweoIG/TaskForces/CommunityProjects/LinkingOpenData

Karwayun, 2010). Several semantic reasoners have been developed to reason over the emerging Semantic Web knowledge. Some of them are reviewed below, with a comparison presented in Table 2.4 based on Singh and Karwayun's comparative study (2010). The first three are Rete-based rule engines. However, none of them support reasoning over data streams.

BaseVISor (Matheus et al., 2006) is a Rete-based, forward-chaining rule engine optimised for processing RDF triples. It supports RuleML and R-Entailment rules. R-Entailment is a language that combines RDF, RDFS and a part of OWL DL with simple Horn-style rules. BaseVISor works similarly to other Rete-based inference engines. However, there is a major difference between BaseVISor and these engines, which is that BaseVISor applies a simple data structure to its facts rather than the arbitrary list structures used by other engines, which enhances the pattern matching efficiency. BaseVISor is implemented in Java and it provides an API to facilitate the addition of user-defined procedural attachments.

Bossam (Jang and Sohn, 2004) is another Rete-based, forward-chaining reasoner for inferencing and querying over RDF(S) and OWL data sets, as well as executing rules such as SWRL. Bossam is based on Logic Programming and First-Order-Logic (FOL). It also provides an API for controlling the engine, loading ontologies and rules, querying RDF(S)/OWL documents and giving explanations about derived facts. However, it does not support SPARQL queries and it does not serialise the knowledge to a persistent store (Papataxiarhis et al., 2009).

Jess (Friedman-Hill, 2002) is a rule engine written in Java that was inspired by the CLIPS project (Wygant, 1989). It uses LISP-like syntax for its rules. It also contains a scripting environment which makes it become a Java framework. Furthermore, being a Java-based system facilitates Jess's integration with a number of Web programming paradigms, like Java servlets or applets. Finally, it supports backward-chaining and some additional features such as procedural attachments (Papataxiarhis et al., 2009).

Jena (McBride, 2002) – a java framework for building Semantic Web applications – provides a rule-based inference engine. However, Jena's RDFS reasoner does not support data types and blank node entailments. Additionally, Jena's OWL reasoner is very limited, since it is a rule-based implementation of OWL-Lite. However, Jena is able to be connected to most of the available Description Logic reasoners, as external reasoners. RacerPro – the commercial extension of Racer – (Haarslev and Muller, 2001), Pellet (Sirin

et al., 2007), and FaCT++ (Tsarkov and Horrocks, 2006) support OWL-DL reasoning using similar tableau-based approaches. However, FaCT++ does not support rules, i.e. ABox reasoning. While all of these reasoners perform consistency checking, Pellet can also explain the reasons for inconsistency.

Table 2.4: Comparison of semantic reasoners

	OWL-DL Reasoning?	Supported expressivity for reasoning	Reasoning method	Rule support?
BaseVISor	No	R-entailment, OWL 2 RL	Rule-based	Yes, SWRL and RuleML supported
Bossam	Yes	LP and negation, incomplete OWL DL	Rule-based	Yes, SWRL supported
Jess	Yes	Not clear	Rule-based	Yes, SWRL supported
Jena	No built-in OWL-DL reasoner	Incomplete for complex description logics	Rule-based	Yes
RacerPro	Yes	SHIQ	Tableau	Yes, SWRL supported
Pellet	Yes	SROIQ	Tableau	Yes, SWRL supported
FaCT++	Yes	SROIQ	Tableau	No

2.3 Conclusion

In this chapter, we discussed the main technologies that have contributed to the emerging semantic stream processing field. A review of the main techniques for data stream processing was presented first, followed by a review of the Semantic Web and reasoning techniques. The next chapter reviews the literature in the specific area of semantic stream processing that combines features of the fields overviewed here.

Chapter 3: Semantic Stream Processing

The relatively new research area of stream reasoning was first identified in 2008 by Della Valle et al. (2008). It aims to integrate data streams with reasoning techniques to enable logical reasoning on real time, semantic data streams. The area evolved fast in the last few years, and is recently referred to as RDF stream processing. In the Semantic Web, data is represented in RDF, and queries can be performed using SPARQL. However, to express the temporal nature of streaming data, RDF needs to be extended to represent time, which is an important concept in data streams. SPARQL also cannot support queries on streaming data as it lacks crucial operators found in the data stream management systems, such as the window operators; this is a result of SPARQL's origins as a store-and-query language. Other issues in this area are: reasoning, distribution, publishing, etc. The following subsections review the existing literature that addresses these challenges within the semantic stream processing area.

3.1 Processing RDF Streams

To enable the processing of RDF streams, Semantic Web research has been extended in two dimensions: support of the representation of time-varying data to enable time-aware processing of such data and support for continuous queries to process streaming linked data on the fly. Several RDF stream processing systems have been developed including C-SPARQL (Barbieri et al., 2010a), Streaming SPARQL (Bolles et al., 2008), $\text{SPARQL}_{\text{Stream}}$ (Calbimonte et al., 2010), EP-SPARQL (Anicic et al., 2011), CQELS (Le-Phuoc et al., 2011) and INSTANS (Rinne et al., 2012a). They generally extend the RDF data model with time annotations to represent data streams and extend the SPARQL query language with streaming operators to enable continuous queries. However, because they differ in syntax and operational semantics, Dell'Aglio et al. (2014) proposed the RSP-QL as a unifying query model. We briefly describe these RDF and SPARQL extensions and discuss each work in greater detail.

In terms of representing data streams, research in this area has followed the conventional definitions of relational data streams as unbounded sequences of data items ordered by timestamps, as used by the DSMS community. All of the above-mentioned systems use the notion of RDF statements as the data items of which the stream consists. With the exception of INSTANS, where the time dimension is implicit in the schema, all other

systems use explicit time instants associated with each RDF statement. The time instants in RSP-QL, C-SPARQL, SPARQL$_{Stream}$ and CQELS are represented as a single timestamp, where the associated statement occurs. In contrast, Streaming SPARQL and EP-SPARQL use an interval of two timestamps to represent the validity of the associated statement. While both ways allow time-aware processing of data streams, using intervals can support richer forms of temporal operations. For instance, in a system that assigns validity to events in the form of time intervals, users can query for events that overlap, or for an event that happens during another event. Beyond the basic RDF stream definition, RSP-QL also formally defines an instantaneous RDF graph, which is a snapshot of the input data taken at a specific point in time that captures the changes of an RDF graph over time.

While different systems propose different extensions of SPARQL to enable continuous querying of data streams, they generally tend to follow the relational CQL model (C-SPARQL, SPARQL$_{Stream}$, CQELS, RSP-QL) reviewed in the previous chapter, Section 2.1.2. They adapt CQL's three classes of operators to work on RDF streams. The first class, that of window operators that transform streams to relations, is adapted to support transforming RDF streams to solution mappings. Most existing works in this area (e.g., C-SPARQL, EP-SPARQL, SPARQL$_{Stream}$, CQELS) support a time-based sliding window, and some also support triple-based sliding windows. The second class of operators (relation-to-relation) includes operators that represent SPARQL algebra (Pérez et al., 2006). Therefore, the main advantage of the CQL approach is that the streaming extension of the query language does not need to redefine its original operators; it only works on their inputs and outputs. The third class (relation to stream) transforms the mapping set produced by the previous class into an RDF stream. The C-SPARQL supports Rstream, while CQELS supports Istream and SPARQL$_{Stream}$ supports Rstream, Istream and Dstream.

Execution strategies vary considerably between different systems. For instance, C-SPARQL, Streaming SPARQL and SPARQL$_{Stream}$ update their input windows periodically, while EP-SPARQL, CQELS and INSTANS use a data driven approach. When a window is updated, systems generally re-execute the standing query on the new window content. In terms of implementation, they tend to use existing engines to process data, where the RSP engine works as a wrapper and orchestrator. As an example, C-SPARQL is built on top of the STREAM DSMS (reviewed in Section 2.1.1) and Sesame SPARQL query engine.

Next, we will discuss the existing works in this area individually and in greater detail, highlighting differences to the above general model.

Streaming SPARQL. The first attempt to extend SPARQL to support processing of RDF streams was presented by Bolles et al., (2008). They introduced Streaming SPARQL as a SPARQL extension to cope with window queries over RDF streams. The definition of RDF stream data type is based on Krämer and Seeger (2005), where three types of streams are defined: a raw data stream represents the data received by the engine, a physical data stream which can be processed by the system operators, and a logical data stream over which the semantics of the extensions can be defined. Instead of performing a window operation on a stream to transform it into a relation, Streaming SPARQL extends the logical SPARQL algebra on the foundation of a temporal relational algebra based on multi-sets, and provides an algorithm to translate SPARQL queries into the new extended algebra. Streaming SPARQL has both tuple based (SlidingTupleWindow) and time based (SlidingDeltaWindow and FixedWindow) window operators. Window operation can be specified in the FROM clause of the query and also in the GroupGraphPattern part.

C-SPARQL. Continuous SPARQL (Barbieri et al., 2009; Barbieri et al., 2010a) is usually considered as the leading contributor in this area and is often cited as a reference in the field (Margara et al., 2014). It is a SPARQL extension that follows a CQL-like approach. It defines an RDF stream data type, adds support for windows over streams and aggregation capability. An RDF stream is simply defined as an ordered sequence of pairs, each of which is made of an RDF triple and a timestamp. As these data streams are possibly unlimited, C-SPARQL has defined two types of windows. Physical windows can extract a given number of triples starting from the last element, while logical windows are time based and can be sliding or tumbling. In C-SPARQL syntax, windows are defined as part of the FROM clause of queries. In addition, C-SPARQL offers more expressivity by defining a number of aggregate functions: count, sum, average, min, and max. While it does not add explicit temporal operators, C-SPARQL allows queries to directly access timestamps of individual triples.

An execution environment for C-SPARQL is also introduced, where each query is translated into two parts: the static part is passed to a SPARQL query engine (Sesame (Broekstra, 2002)), while a relational data stream management system (STREAM (Arasu et al., 2003)) is used to evaluate the streams and aggregates. Although this framework has the advantage of reusing existing technology, splitting queries into static and dynamic parts may prevent optimisation that could be possible in a unified framework.

EP-SPARQL. Event Processing SPARQL (Anicic et al., 2011) is another SPARQL extension proposed as a new language for event processing and stream reasoning. Unlike

other systems that are inspired by Data Stream Management Systems, EP-SPARQL's language constructs and processing model is based on CEP systems. EP-SPARQL is defined to be SPARQL extended by the binary operators SEQ, EQUALS, OPTIONALSEQ, and EQUALSOPTIONAL. These operators are used to detect RDF triples occurring in a specific temporal order in order to capture more complex patterns over RDF streams. These expressive temporal operators go beyond C-SPARQL's simple timestamp() function in terms of detecting temporal relationships between RDF patterns. Moreover, EP-SPARQL provides the function getDURATION() to add selection criteria (windows) over RDF streams, which are defined as sequences of RDF triples where each triple is associated with a time interval.

EP-SPARQL queries are translated into the Etalis language for events (ELE) (Anicic et al., 2012) rules. Etalis is based on logic programming and implemented in Prolog. Etalis rules are compiled as event-driven backward-chaining (EDBC) rules, which enable event-driven, incremental detection of complex events in near real time. By using (recursive) logic rules, a unified execution mechanism for both querying and reasoning is enabled.

SPARQL$_{Stream}$. Calbimonte et al., (2010) designed a service that enables ontology-based access to streaming data in order to integrate heterogeneous data sources. The service receives queries specified in terms of the ontology using SPARQL$_{Stream}$, another extension of SPARQL, to support processing RDF streams. These queries are then transformed into a relational continuous query language (SNEEql) using a set of mappings expressed in S2O, an extension of the R2O mapping language (Barrasa et al., 2004) that supports streaming data. After the query translation phase, the query processing phase starts, using a DSMS engine (SNEE) (Galpin et al., 2011). The results are then transformed from a set of tuples into ontology instances. SPARQL$_{Stream}$ language is inspired by – and very similar to – C-SPARQL. However, it adds support for the window-to-stream operators: Rstream, Istream, and Dstream, and it only supports time-based windows.

CQELS. As opposed to C-SPARQL, EP-SPARQL, and SPARQL$_{Stream}$, CQELS (Continuous Query Evaluation over Linked Streams) (Le-Phuoc et al., 2011) uses a "white box" approach, i.e. it defines its own RDF-native processing operators rather than reusing existing technologies. Hence, it integrates the processing of background and streaming data without delegating each of them to external engines. CQELS uses a point-based timestamp to add the temporal aspect for both streams and linked data. To process its input, CQELS implements three types of operators, organised as a data flow: window, relational, and streaming operators, that resemble CQL's stream-to-relation, relation-to-relation, and

relation-to-stream operators. CQELS also has the feature of adaptive queries processing, where a mechanism similar to Eddies (Avnur and Hellerstein, 2000) is used to dynamically reorder the operators in the data flow tree. A cost-based routing policy decides the order in which the operators are executed at runtime.

INSTANS. A completely different approach to processing dynamic RDF data is introduced in INSTANS (Incremental eNgine for STANding Sparql) (Rinne et al., 2012a). Instead of extending SPARQL to support continuous queries, they used the Rete algorithm (Forgy, 1982), an incremental algorithm to solve the many patterns/many objects match problem, to implement a number of interconnected SPARQL 1.1 updates. It also differs from standard SPARQL in the execution mechanism because it does not execute queries on demand but rather propagates data through a query matching network. Each tuple is processed as soon as it arrives, and output is produced immediately when all conditions match. It differs from continuous SPARQL extensions in that it has no notion of stream-to-relation or relation-to-stream operators; instead, windows are handled using explicit INSERT and DELETE queries. Therefore, it is possible to use only SPARQL without any extensions to process streams. However, we can show that this comes at the expense of a more complicated way to model the required query semantics, as follows. In Rinne et al. (2012b), in a sample use case in which one needs to detect a nearby friend, which can be represented in C-SPARQL as a single continuous query, four SPARQL queries are needed in INSTANS: a window query, a nearby detection query, a notification query and a query to remove invalidated nearby statuses.

Table 3.1 compares the different approaches to enabling RDF stream processing; the execution is mainly based on Le-Phuoc (2012a). Systems use similar data models and provide a similar level of expressivity, except for EP-SPARQL, which supports higher levels of expressivity; however, they vary in their execution models. Two important points can be observed from the table. First, none of them, except EP-SPARQL, support background ontological reasoning. Even in EP-SPARQL, ontological reasoning is not supported natively but rather enabled using user-defined recursive production rules. The next section discusses works that are mainly focused on the reasoning problem. Second, we notice that systems either use static plans or completely externalise the optimisation problem to the underlying systems. This is mainly because using a black-box approach prohibits optimisation opportunities. CQELS, which enables native support of RDF streams, is the only engine to provide adaptive optimisation.

Table 3.1: Comparison of RDF Stream Processing approaches

		Streaming SPARQL	C-SPARQL	EP-SPARQL	$SPARQL_{Stream}$	CQELS	INSTANS
Data model	**Stream elements**	Triples	Triples	Triples	Triples	Triples	Triples
	Time model	Interval	Single point	Interval	Single point	Single point	Implicit
Expressivity	**Supported windows**	Time-based, triple-based	Time-based, triple-based	Time-based	Time-based, history windows	Time-based, triple-based	No explicit window operator
	Temporal operators	No	No	Yes	No	No	No
	Reasoning	No	No	RDFS subset	No	No	No
Execution	**Input updates**	Periodical	Periodical	Data-driven	Periodical	Data-driven	Data-driven
	Execution strategy	Re-execution	Re-execution	Incremental	Re-execution	Re-execution	Incremental
	Optimisation	Algebraic, static	Algebraic, static	Externalised	Externalised	Physical, adaptive	Algebraic, static
	Internal representation	Stream-to-stream operators	CQL&SPARQL queries	Logic programmes	SNEE queries	Adaptive physical operators	Rete dataflow networks
	Underlying engine	N/A: No implementation	Sesame and STREAM	Prolog	SNEE	N/A: Native	N/A: Native

3.2 Reasoning on Semantic Streams

While the RSP engines enabled processing RDF streams, integration with static knowledge bases and issuing of standing SPARQL queries, RDFS/OWL reasoning was not supported by the majority. Other works in the area addressed the reasoning problem in different ways. We classified the proposed approaches towards stream reasoning depending on their expressivity in two categories. The first supports lightweight reasoning to highly dynamic streams on the level of RDFS and subsets of OWL 2 RL. The second enables richer forms of reasoning, including description logics and nonmonotonic reasoning; however, it either does so for less dynamic streams or involves approximation techniques. Stuckenschmidt et al. (2010) introduced a conceptual view of cascading reasoners, in which reasoners are organised in a hierarchy of increasing complexity in order to overcome the trade-off between the complexity of the reasoning method and the frequency of the data stream the reasoner is able to handle.

3.2.1 Lightweight stream reasoning

An early work by Walavalker et al. (2008) presents a subsumption reasoner that can deal with streaming knowledge. Given an RDFS or OWL ontology, the system pre-computes the transitive closure of all classes on the rdfs:subClassOf relationship and stores the class-subclass pairs in a database table. A set of continuous queries are defined based on the RDFS entailment rules, to be evaluated at run time by the TelegraphCQ stream management system (Chandrasekaran et al., 2003), in order to identify subclass events of the event of concern. Similar to C-SPARQL, EP-SPARQL, and $\text{SPARQL}_{\text{Stream}}$, this reasoner uses a black-box approach. Its expressivity is limited to the main RDFS predicates (subClassOf, subPropertyOf, range, and domain) and OWL's inverseOf relationship.

Reasoning over streams needs to operate incrementally, as re-computing all results whenever a change is made to the input streams (insertion or deletion) can result in a very slow performance. Barbieri et al. (2010b) propose an approach for incremental reasoning to maintain the ontological entailments, referred to as IMaRS in (Dell'Aglio and Della Valle, 2014). The approach is based on the (DRed) algorithm (Gupta et al., 1993), which overestimates the deletions and then computes re-derivations, but the resulting incremental algorithm of data streams reasoning is easier and more efficient because the addition or removal of facts from data streams is controlled by windows, which have a clear expiration

time. The algorithm requires tagging of each RDF triple (both inserted and entailed) with an expiration timestamp. The program then can compute a new complete and correct materialisation by dropping RDF triples that are no longer in the window. This can be directly compared to the Rstream operator of CQL (Arasu et al., 2003), which provides all answers that are correct at a certain time as a stream of results. An evaluation of the program shows that it is faster than the naïve approach (computing the entire materialisation at each step) when the percentage of change is less than 13% of the background knowledge. Although the results are promising, the experiments have been performed for only one query, so it is not clear how the algorithm will work and scale for multiple queries with different window definitions (Anicic et al., 2011). Furthermore, it is restricted to time-based windows and does not allow deleting triples, e.g. in case of inconsistency, before their expiration.

The Rete algorithm also works incrementally. Rete networks are used in Sparkwave (Komazec et al., 2012) to enable schema-enhanced pattern detection on RDF data streams. However, they present a fixed approach that can only operate over RDF schema and a few OWL constructs. A pre-processing epsilon network, which handles the reasoning task, is placed before the Rete network, which processes the RDF streams. Sparkwave is a clear example of sacrificing expressivity for performance; its reasoning capabilities are restricted, it has the same limitations as IMaRS and it does not support disjunction, negation, temporal or arithmetic operators. Oliya et al., (2011) also use the Rete algorithm to enable incremental OWL reasoning over dynamic contextual information that can be expressed through Description Horn Logic ontologies. However, the paper says nothing about time windows or any other streaming operators.

3.2.2 Complex stream reasoning

Rscale (Liebig and Opitz, 2011) is an OWL 2 RL reasoner that is suitable for a moderate update frequency. The system uses a relational database as secondary storage to store the ontology in base tables. The applicable rules are translated to SQL queries and executed by the rule engine at those tables. The results are written in delta tables, followed by an alignment and merge phase, which executes in rounds by deleting already inferred facts from delta tables and adding the remaining facts to the base tables. Dynamic updates (insertions and deletions) are then dealt with incrementally based on Volz et al. (2005). Because the execution mechanism consists of many steps and requires reactivation with

every update, Rscale might not be suitable for fast streams. Furthermore, it does not support time-aware reasoning.

On a more expressive level, TrOWL (Ren et al., 2010; Ren and Pan, 2011) supports dynamic management of description logic ontologies. It uses syntactic approximation to reduce reasoning complexity. In contrast to IMaRS, TrOWL requires no fixed time window to manage deletions; instead, it uses a Truth Maintenance System to maintain intermediate results and the deduction relations among them. Because this approach has the disadvantage of using excessive memory consumption, they also present an optimisation algorithm that reduces the number of unnecessary intermediate results.

Do et al. (2011) introduced the concept of stream reasoning to Answer Set Programming (ASP) using dlvhex (Leone et al., 2006) to combat uncertain data by using disjunction rules to generate a multiple answer set. They applied an ASP solver (dlvhex) repeatedly on periodically changing windows of OWL objects. On the contrary, Gebser et al. (2012) handled streaming data into the reasoning methodology of ASP by proposing novel language constructs that enable specifying and reasoning with time-decaying logic programs. Similar to the idea of cascading reasoners (Stuckenschmidt et al., 2010), StreamRule (Mileo et al., 2013) combines CQELS for stream processing and filtering with Oclingo, the ASP engine of Gebser et al. (2012), for nonmonotonic reasoning.

On the theoretical side, LARS (a logic-based framework for analysing reasoning over streams) (Beck et al., 2015) provides a rule-based formalism with different means for time abstraction. A rule language with model-based, nonmonotonic semantics similar to ASP is also introduced.

3.3 Publishing Semantic Streams

Though best practices for linking semantic static data on the Web were stated under the name of Linked Data (Bizer et al., 2009), RDF streams have been neglected. Publishing RDF streams in the Linked Data cloud (Abele et al., 2017) will enable Semantic Web applications to consume this data and facilitate the integration of RDF streams with static data already published in the Linked Data cloud. To enable such publication, the identification, discovery, and access to stream data need to be addressed (Sequeda and Corcho, 2009).

The concept of Linked Stream Data has been introduced by Sequeda and Corcho (2009). They propose a URI-based mechanism to identify and access sensor network streams. Sensors are identified by URIs that return the sensors' metadata when dereferenced, while data streams emitted by those sensors are also identified by URIs that return the observations contained in the stream. The Linked Stream Data URI scheme also identifies stream data at specific moments in time (by including a timestamp in the stream URI), in specific time windows (by specifying a start time and end time in the stream URI), and at specific locations (by including the coordinates in the stream URI).

Another approach to publish data streams as Linked Data was proposed by Barbieri and Della Valle (2010c). In this approach, each RDF stream is represented as one Stream Graph (s-graph) and several Instantaneous Graphs (i-graphs). A s-graph is a metadata graph that describes the current content of the window over the RDF stream. S-graphs use the rdfs:seeAlso attribute to refer to a number of i-graphs, which in turn represent individual readings or observations. Similar to the Linked Stream Data approach, these s-graphs and i-graphs are identified by URIs. However, representing time is different. While Linked Stream Data allows for opening a window starting from and ending at any moment in time, this approach specifies only the duration (or size) of the window in the URI, forcing the extraction of the last elements from the data stream. This is more compliant with the nature of streams, that being possibly of unbounded size, should not be treated as persistent data to be stored and queried on demand, but rather as transient data to be consumed on the fly by continuous queries.

3.4 Distributed Semantic Stream Processing

Systems that deal with a high velocity or volume of data streams feature several scalability requirements (Shah et al., 2003). These requirements have been addressed in relational DSMSs by enabling parallel and distributed processing of data streams (e.g. Abadi et al., 2005; Shah et al., 2004). In the stream reasoning area, there is an attempt by Hoeksema and Kotoulas (2011) to apply a parallel approach for stream reasoning using the Yahoo S4 framework. They introduce a number of RDFS specialised reasoning Processing Elements (PEs) to distribute triples over multiple streams. Streams are distributed across those PEs according to a given key, so the PEs can be distributed across different nodes for a parallel execution. Continuous query answering is also supported by a number of components that can be combined to translate a subset of C-SPARQL into a parallel execution plan.

While all the RSP engines reviewed in sections 3.1 work in a centralised setting, the recent implementation of CQELS (Le-Phuoc et al., 2013) supports parallel processing in a cloud environment. They built the CQELS engine on an elastic cluster, where it can adapt to changing processing loads by dynamically adjusting the number of processing nodes at runtime. The network consists of a number of processing nodes and a central coordinator that maps the logical query network to the available nodes. They provided parallel algorithms for window, join and aggregation operators. Their evaluation experiments show that the throughput scales linearly with increasing numbers of processing nodes.

3.5 Developed Semantic Streams Environments

SemSorGrid4Env (Gray et al., 2011) is an application that implements a service architecture to provide a semantically integrated information space for sensed and stored data drawn from heterogeneous data sources. The architecture is structured into three tiers. The data tier enables publishing and querying data in its native format. The middleware tier supports the discovery and integration of different data models. Finally, the application tier provides support for web-based applications to interact with other services of the system. This architecture is deployed as a flood emergency response planning system. The paper mainly focuses on the integration process, while they use $\text{SPARQL}_{\text{Stream}}$ (Calbimonte et al., 2010) for continuous query processing.

BOTTARI (Balduini et al., 2012) is a mobile application that continuously analyses social media streams to deliver personalised location-based recommendations. BOTTARI architecture contains three parts: a client (mobile app) which interacts with the user and initiates SPARQL queries, a PUSH segment that continuously analyses streams of tweets using C-SPARQL, and a PULL segment that answers the client's queries by combining different forms of reasoning. The paper presents an evaluation of a deployment of the system that analyses tweets about points of interest (POIs) such as hotels and restaurants in Insadong, Korea. Their scalability test shows that the system handles a flow of 15,000 tweets/second, though the input rate is at tens of tweets a day. As this rate is considered slow for a stream processing engine, they claim that BOTTARI's scalability goes largely beyond the actual needs of its deployment in Insadong.

Another platform is called Linked Streams Middleware (LSM) (Le-Phuoc et al., 2012b). It integrates sensor streams with Linked Data by enriching them with semantic descriptions using a wide range of wrappers. It also provides an intuitive Web interface for data

annotation and visualisation. Finally, live querying over both types of data is enabled using CQELS and a standard SPARQL processor. LSM uses a cloud-based infrastructure (Hadoop cluster) for real-time data collection, which enables its current deployment to access over 110,000 sensor data streams. However, the continuous querying process is not distributed.

3.6 Benchmarking

As the number of RDF stream processing engines has increased, the need for an open benchmarking framework has grown. Existing RDF/SPARQL benchmarks such as the Berlin SPARQL Benchmark (Bizer and Schultz, 2009) and LUBM (Guo et al., 2005) are designed for static data. On the other hand, there is also an available benchmark for DSMSs: the Linear Road benchmark (Arasu et al., 2004), which is based on the relational data model, and does not consider reasoning. Therefore, a number of benchmarking frameworks for streaming RDF/SPARQL engines have been proposed.

The Streaming RDF/SPARQL Benchmark "SRBench" (Zhang et al., 2012) is mainly focused on evaluating coverage for SPARQL constructs. It uses a real world sensor data set linked to some static data sets from the LOD cloud, and provides a comprehensive set of queries that cover the important SPARQL operators and the common streaming SPARQL extensions. They provided a functional evaluation of three streaming SPARQL engines: $\text{SPARQL}_{\text{Stream}}$, CQELS, and C-SPARQL. The results show that all three engines support basic SPARQL features over time-based windows of streaming data. None of the engines provides reasoning, and there is very limited support for SPARQL1.1 features. At the moment, SRBench does not offer performance evaluation.

The second benchmark is the Linked Stream Benchmark "LSBench" (Le-Phuoc et al., 2012c) focuses on evaluating performance of the system using throughput as an indicator. LSBench implements a data generator that generates stream social network data. It also defines a set of queries to test the functionality of three streaming engines: CQELS, C-SPARQL, and JTALIS (EP-SPARQL engine). The correctness of the results of these queries is then tested, which shows there is a degree of mismatch due to the different execution strategies used by the engines. LSBench also provides some performance tests measuring throughput of the three engines to see how fast they are. As C-SPARQL is not designed for large static data sets, the results show that CQELS and JTALIS have higher throughput than C-SPARQL by some orders of magnitude. The same test is run again with

varying static data sizes, and different number of simultaneous queries. CQELS outperforms the other engines but they all show linear deterioration of throughput against increasing numbers of queries. LSBench does not measure other performance metrics, such as memory usage.

An extension of the SRBench that is mainly concerned with the correctness problem is called CSRBench (Dell'Aglio et al., 2013). When comparing the output of different RSP engines, it is difficult to determine whether different results for the same query from different engines are incorrect behaviour or merely a result of different operational semantics. To address this problem, they analyse the operational semantics of these engines, focusing on stream-to-relation and relation-to-stream operators. Stream-to-relation analysis is based on the SECRET (Botan et al., 2010) model, which characterises the behaviour of time windows through four functions: scope, content, report and tick. For relation-to-stream semantics, they consider the operator used (Rstream, Istream, or Dstream) and whether or not the engine produces a notification for an empty answer. However, their approach is only applicable to CQL-based systems.

Another framework for benchmarking RSP engines is CityBench (Ali et al., 2015), which focuses on evaluating system performance under a realistic dynamic setting in the smart city domain. It provides a configurable testbed infrastructure, which allows the use of evaluation tests using fine-tuned configuration parameters. These include changes in input streaming rate, variable background data size, number of concurrent queries and number of streams within a single query. They evaluated two RSP engines (C-SPARQL and CQELS) in terms of latency, memory consumption and completeness of results while varying the configurable parameters.

3.7 Conclusion

The processing of dynamic data has gained increasing attention in the Semantic Web community over the past few years. Research efforts have addressed several issues in this area including defining and querying semantic streams, stream reasoning, publishing linked streams, and benchmarking semantic stream processing systems. Semantic streams are usually defined as sequences of RDF triples associated with a time element. However, there is a room for other definitions of RDF streams based on different granularity, e.g. streams of RDF graphs.

The main focus in the area was to enable on-the-fly processing of semantic streams, which led to the development of a number of stream processing engines that can handle dynamic RDF streams. However, apart from CQELS, all of these engines are not natively designed for RDF streams. CQELS defines its own operators to work directly on RDF streams. However, it does not support reasoning.

SRBench (Zhang et al., 2012) found that none of the tested stream processing engines provides reasoning support. The few studies that aim to provide streaming inference support have been reviewed in this chapter. Areas such as real-time optimisation and distributed processing of RDF streams remain very little explored.

In conclusion, there are some RDF stream processing systems (Table 3.1) but the majority of them do not support reasoning. On the other hand, there are some stream reasoning systems (reviewed in Section 3.2) but they do not address the optimisation problem. Therefore, our work aims to close this gap, by providing reasoning support as well as adaptive optimisation in a native, unified approach.

Chapter 4: Continuous Reasoning

While most of the leading works in semantic stream processing area focus on RDF stream processing without reasoning support (Zhang et al., 2012), we mainly aimed to enable reasoning for RDF streams. In this chapter, we describe our approach to support generic rule-based reasoning for real-time streaming data. First, we define a number of general requirements that informed our design decisions. Second, we introduce our continuous reasoning framework, including a definition of the RDF stream datatype, and an operational description of the supported operators. Then, we introduce R4 – Rule-based Reasoner for RDF streams using Rete – our prototype stream reasoning engine. We describe its architecture, how it implements the different operators of the framework, and present an extension to the standard Semantic Web rule language RIF in order to add support for temporal operations.

4.1 Requirements

Different scenarios have different requirements, mainly due to different stream characteristics (input rate, volume, bursts, etc.), resource constraints, and the nature and complexity of the rules to be applied. For example, network analysis systems must deal with high-volume input streams (Sullivan, 1996), while sensor networks should tolerate the power constraints of the sensors (Akyildiz et al., 2002). This section presents some common requirements and challenges that should be considered to enable the processing of semantic data streams. Most of these requirements are based on Stonebraker et al., (2005) who defined general requirements for stream processing applications (DSMSs), with the addition of inference support that is only relevant to semantic streams.

R.1. Integration

In many cases, it is essential to combine stored data with streaming data to provide answers to queries. For example, names and locations of sensors are typically static stored data, while the sensors' observations are streaming data. For instance, queries asking for the 'number of cycles hired in the last ten minutes in central London' require a combination of both stored and streaming data to be evaluated. It is important to have an efficient mechanism for the unified processing of stored and streaming data. Furthermore, many scenarios call for the integration of data from different sources (Margara et al., 2014),

which is the main reason for modelling data in the RDF format that enables interoperability.

R.2. Time management

As streaming data change over time, scenarios in streaming applications are usually concerned with monitoring this change. For example, weather monitoring applications can use changes in meteorological data observed during specific windows of time to predict severe weather conditions. This means that a data model linking the streaming data element to the time domain is needed. In addition to the time-based data model, a time-aware processing model is also needed to model time relations between data.

R.3. Data-driven processing

Some applications – especially in the health and environment domains – are time-critical, demanding fast response times. For example, a home health monitoring application (Paganelli and Giuli, 2007) should alert the hospital when it infers that a critical situation has occurred based on abnormal biomedical parameters observed by its sensors. In general, answers should be provided before they become outdated or useless. To achieve this, latency[10] should be kept to a minimum. In a data-driven environment, stream elements are consumed and results are generated as soon as they are received. Stonebraker et al., (2005) describes data-driven systems are described as 'active' systems, as opposed to 'passive' systems, which wait to be told what to do by an application before beginning processing.

R.4. Memory utilisation

As data streams are potentially unbounded in size, complete processing of the whole dataset would also require unbounded memory. To address this challenge, quality and completeness of results can be traded for memory space.

R.5. Inference support

One of the distinctive features of the Semantic Web is expressive inference capabilities. Applications that model their streaming data using Semantic Web languages should support some forms of reasoning, as some queries need implicit knowledge. For example, in the social media domain, a possible scenario (similar to scenarios in (Balduini et al., 2013)) is to request recent tweets with hash tags related to a specific city. Reasoning over an ontology that models the different districts in this city can retrieve these tweets. As reasoning over expressive ontologies is considered expensive, there is a trade-off between expressivity and efficiency.

[10] By latency, we refer to response time, the time between the arrival of new elements and the generation of output results. We use both terms interchangeably.

R.6. Managing dynamic environments

Response time and memory consumption are highly dependent on the internal processing plans generated by systems. Implementing a good optimisation strategy can improve the system's performance. However, during the life of the running plan, the environment conditions (e.g. stream's input rate) may change, so the system needs to support adaptive optimisation (Babu and Bizarro, 2005). For example, in a street monitoring application, input rates can vary significantly between busy and quiet periods. Without adaptivity, performance may drop significantly as conditions change over time. This is because queries and rules in a streaming environment are long-running, so a bad optimisation plan causes long-term damage to performance.

Relation to other requirements in the literature: A recent survey of the stream reasoning area (Margara et al., 2014) analysed a number of scenarios and produced requirements similar to those mentioned above. These requirements include integration (R.1); time management (R.2); efficiency, which is directly related to the low latency requirement (R.3); big data management, which includes memory utilisation (R.4) and managing dynamic environments requirements (R.6); expressivity as a more general form of the inference support (R.5); and Quality of Service, which is also related to managing dynamic environments and adaptivity (R.6). Other requirements that go beyond the scope of this thesis are distribution, uncertainty management, and historical data management. We consider distribution as a future work. While reasoning with uncertainty (Halpern, 2005) is an important requirement, we consider it of secondary importance to the requirements we address.

As stated in the beginning of this section, some of our requirements were inspired by Stonebraker et al.'s (2005) eight rules for stream processing. Their first rule is to 'keep the data moving', which is related to our low latency requirement (R.3). Their second rule is to 'query using SQL on streams'. Although querying using SQL might be irrelevant for linked streaming data, this requirement demands support for time-aware data processing, which is related to our time management requirement (R.2). Another rule is to 'integrate stored and streaming data', similar to our first requirement (R.1). Furthermore, the 'process and respond instantaneously' talks about the importance of highly optimised plans for achieving good performance, which is related to our requirement regarding adaptive optimisation (R.6). Stonebraker et al.'s other rules are mostly related to distribution and managing uncertainty, including automatic partitioning and scaling, guaranteeing data

safety and availability, handling stream imperfections, and generating predictable outcomes, which we consider out of the scope of this thesis.

Addressing these requirements in the literature: Table 4.1 shows how the most relevant systems in the literature addressed the previous requirements. All of those RDF stream processing systems enable integration of different data sources by processing streams encoded in RDF and also enable integration with static datasets. However, the majority of these systems rely on underlying stream or event processors and SPARQL engines which means a unified processing of stored and streaming RDF data is not enabled (Le-Phuoc et al., 2011). A time-based model for RDF data is supported by most systems by annotating data elements with their time of occurrence or validity. EP-SPARQL also supports temporal reasoning. Data-driven processing is supported by EP-SPARQL, CQELS, INSTANS and Sparkwave. A common approach to utilise memory followed by the majority of these systems is to evaluate queries over sliding windows based on the assumption that users are mainly interested in recent data. Reasoning over a subset of RDFS is enabled in EP-SPARQL and Sparkwave, while IMaRS also supports transitive property. Managing dynamic requirements is only supported by CQELS by enabling adaptive optimisation of the continuous SPARQL queries.

Table 4.1: Addressing the requirements in the related RDF stream processing systems

Systems \ **Requirements**	R.1 Integration	R.2 Time management	R.3 Data-driven processing	R.4 Memory utilisation	R.5 Inference support	R.6 Managing dynamic environments
Streaming SPARQL (Bolles et al., 2008)	*	*		*		
C-SPARQL (Barbieri et al., 2009)	*	*		*		
EP-SPARQL (Anicic et al., 2011)	*	*	*	*	*	
$SPARQL_{Stream}$ (Calbimonte et al., 2010)	*	*		*		
CQELS (Le-Phuoc et al., 2011)	*	*	*	*		*
INSTANS (Rinne et al., 2012a)	*		*	*		
IMaRS (Barbieri et al., 2010b)	*	*		*	*	
Sparkwave (Komazec et al., 2012)	*	*	*	*	*	

4.2 Continuous reasoning framework for RDF streams

A framework that supports the requirements in the previous section in the semantic streams context is needed. The main features of this framework and their relations to the above requirements are listed below.

1. **Unified and native RDF support:**

The majority of the systems reviewed in the previous chapter that enable semantic stream processing (Barbieri et al., 2010a; Calbimonte et al., 2010; Anicic et al., 2011; Walavalker et al., 2008) employ a black-box approach, in which they rely on stream processing engines that are not optimised for the RDF data model. This also introduces the overhead of wrapping or translating the queries and the RDF stream elements to the underlying engine's query and data model (Le-Phuoc et al., 2011).

While almost all RDF stream processing systems enable the integration of streaming and static RDF data, processing these two types of data is not unified, which hinders possible optimisations. In order to offer maximum optimisation opportunities – which have a major impact on response time, memory consumption, and completeness of results – the system should have full control over the low level processing operators. Our framework is based on a white-box architecture, in which processing operators work directly on RDF data streams and background knowledge in a unified approach. This feature addresses the integration requirement (R.1) and is also related to the low latency requirement (R.3).

2. **Continuous reasoning:**

Our processing framework is not only aimed at the continuous processing of high-throughput RDF streams but also adds inference support. The initial main focus of research in this area was to enable the rapid processing of RDF streams. As covered in Section 3.1, only EP-SPARQL adds simple forms of reasoning. As opposed to approaches covered in Section 3.2.2, which support higher complexity reasoning for low-throughput streams, our approach enables lower complexity, general rule-based reasoning at the level of OWL 2 RL on high-throughput streams.

Our reasoning framework includes a time-annotated data model and a time-aware continuous processing model based on time windows. The model is fully streaming so that even inference results are considered streams, themselves, and can enter the system again as input and contribute to derive more results. The continuous reasoning model addresses the inference requirement (R.5) and time management requirement (R.2).

3. **Incremental, data-driven processing**

All operators in our model work in a data-driven manner. Stream elements received from outside streams are directly pushed to the operators, and the operators, themselves, communicate their results in a push-based approach. This ensures minimum latency, as opposed to the periodic evaluation approach used in C-SPARQL, Streaming SPARQL, and $SPARQL_{Stream}$, which update the content of the windows for every period of time specified in the query. For example, if the re-evaluation period is specified as five minutes, there will be a minimum latency of five minutes between the first received triple and the results it produces.

Furthermore, as the inference process is computationally expensive, the naïve re-computation approach is not adequate in a streaming context. Thus, we use the naturally incremental Rete algorithm (Forgy, 1982) to implement the continuous reasoning networks. As the Rete algorithm is memory-intensive, we employ a time-based extension to utilise memory consumption. These techniques aim to improve the performance of the system, addressing the data-driven processing requirement (R.3) and memory utilisation (R.4).

4. **Adaptive optimisation:**

As a consequence of the unified white-box approach, optimisation opportunities should be exploited to improve the performance of the system. With the exception of CQELS, other RDF stream processing systems do not focus on the optimisation problem, as it is often externalised to the underlying engine; cross-model optimisation of stream and static data is restricted due to the black-box approach.

As the traditional RDBMS cost-based optimisation approach is not suited for the streaming context due to the absence of statistics beforehand and the dynamic nature of streaming environments, our framework employs adaptive optimisation techniques based on the research in data stream management systems. The adaptive optimiser responds to changes in the conditions of input streams by re-organising operators in the continuously running dataflow networks. The adaptivity feature allows managing dynamic environments (R.6), enabling lower response time (R.3).

Table 4.2 shows the features supported in our framework and their connections with the requirements.

This continuous reasoning framework is realised using data-flow networks, where rules are translated into pipelined, non-blocking physical operators (that work incrementally in a data-driven fashion) representing nodes and edges represent RDF streams that flow between operators, generating results in a continuous manner. Streams are observed through time windows, which enable the time-based maintenance of both stream elements and inference results. In the following subsections, the notion of RDF streams is formally defined, followed by the supported operators in our framework.

Table 4.2: Requirements and design decisions

Design decisions / **Requirements**	Native RDF support	Continuous reasoning	Incremental data-driven operators	Adaptive optimisation
R.1 Integration	*			
R.2 Time management		*		
R.3 Data-driven processing	*		*	*
R.4 Memory utilisation			*	
R.5 Inference support		*		
R.6 Managing dynamic environments				*

4.2.1 Data Model

Using RDF as a unified data model for all data streams can solve the heterogeneity problem found in the IoT area. However, the basic RDF model needs to be extended to express the temporal aspect of data streams by adding a time annotation. Therefore, as in the DSMS systems, streams are sequences (possibly unlimited) of pairs, each of which is formed of a data element and a time element.

4.2.1.1 The data element

All RDF stream processing systems reviewed in the previous chapter use the RDF statement as the data element, defining an RDF stream as an ordered sequence of RDF triples associated with a time element. However, RDF streams can be seen as streams of events that occur in real-life, and while some of these events can be represented using a single statement, in many cases, they need many more triples to convey their meaning. For example, a person entering a room can be represented in one RDF statement, while 14

triples are used to represent a single sensor observation in the SemSorGrid4Env project using the SSN ontology. This set of triples that describes a single event can be considered as an RDF graph.

Defining the stream as a sequence of graphs can also be more meaningful in other situations. First, use cases that require tuple-based windows do not work correctly if events are represented in more than one triple using a triple-based definition of streams. For example, a query asking to observe the last ten tweets of a specific user does not give correct results using a window of size 10 over a triple-based stream, as tweets are usually represented in more than one triple. Second, in systems that apply sampling techniques over input streams for load shedding, sampling at a graph level should produce more results than sampling at a triple level. For example, if a tweet is represented in two triples (e.g. one represents the user; one represents hash tags used), a simple sampling technique that sheds 50% of the load will produce no results if the query concerns both triples of the tweet (e.g. a specific user and a specific hash tag) using the triple-based definition, while sampling on a graph-level can produce 50% of the qualifying tweets.

For these reasons, we define RDF streams as sequences of RDF graphs associated with time annotations. This definition is general enough to capture cases where an event is represented using a single triple, as it can be a special case of a graph containing one statement. This definition represents external streams arriving at the system from a data source. An external stream is an ordered sequence of pairs; each pair consists of an RDF graph and a time element. On the other hand, internal streams represent data that flows between operators inside the system. Our operators work on the fine-grained level of triples. The data element of internal streams depends on the operator that produced it. For example, an output stream of a filter operator (checking a triple pattern) is a sequence of triples, while an output of a join operator is a number of joined triples representing a partial result. These partial results are lists of triples, i.e. graphs. However, we refer to them as tokens in order to avoid confusing the partial results with the external stream graphs. An internal RDF stream is an ordered – possibly unlimited – sequence of tokens associated with a time element.

4.2.1.2 The time element

For the time element, we follow the temporal model of Krämer and Seeger (2005), where external stream elements are associated with timestamps (representing time of occurrence) and internal stream elements are annotated with time intervals (representing validity). If

external graphs arrive at the system without timestamps, they should be stamped by the system (system time) prior to any processing. If they come annotated with application time, we expect them to arrive in order.

Upon entering the system, external streams are transformed to internal streams by adding an expiration time to the occurrence (start) time. The expiration time for each element can be calculated using time window operators by adding the specified window size to the element's start time. This facilitates garbage collection and inference result maintenance, as they will also be assigned a time interval indicating their expiration time. This can further enable coalescing value-equivalent streams with adjacent time intervals.

4.2.1.3 Formal Definitions

Definition 1 (Time): Let T be a discrete time domain with a total order under $\leq$; and let element $t \in T$ be a timestamp. A time interval is defined as $[t_s, t_e)$, where t_s and t_e are both timestamps, and $t_s \leq t_e$.

Definition 2 (RDF triple and RDF graph): Like Cyganiak et al. (2014), we define an RDF triple as follows: Let I denote the set of IRI constants, L the set of literals, B the set of blank nodes, (s,p,o) then denotes an RDF triple and its subject-predicate-object components where $s \in I \cup B$, $p \in I$ and $o \in I \cup L \cup B$. We define an RDF graph, G^{RDF}, as a set of RDF triples: $G^{RDF} = \{(s_1, p_1, o_1), (s_2, p_2, o_2), ..., (s_n, p_n, o_n)\}$.

Definition 3 (External RDF stream): We consider an external RDF stream, S_e, to be a sequence of timestamped RDF graphs, and therefore we define it as a sequence of pairs, $< G^{RDF}, t>$, each of which is comprised of an RDF graph and a timestamp $t \in T$. Like Arasu et al. (2006), we note that there could be zero, one, or multiple graphs with the same timestamp in a stream. However, there should be a finite number of graphs with a given timestamp.

Order and equivalence: Stream elements are ordered by their timestamps. For two stream elements, $<G_1, t_1>$ and $<G_2, t_2>$, we say that $G_1 <_t G_2$ to indicate that $t_1 < t_2$, which means that G_1 precedes G_2 in the stream. However, graphs with the same timestamp have no particular order within in the stream; they can appear in any order and the stream will remain the same. We say that G_1 is timestamp-equivalent to G_2 ($G_1 =_t G_2$) if $t_1 = t_2$. Therefore, a stream B can be equivalent to a stream A if B contains the same graphs as A, and for each $G_1, G_2 \in A$, G_1 must precede G_2 if $G_1 <_t G_2$, or G_1 can appear before or after G_2 if $G_1 =_t G_2$. For example, stream A =

{<a,1>, <b,2>,<c,2>, <d,3>} is equivalent to stream B = {<a,1>, <c,2>,<b,2>, <d,3>}.

Definition 4 (Internal RDF stream): We consider an internal RDF stream, S_i, to be a sequence of tokens associated with time intervals, and therefore we define it as a sequence of pairs, <K, [t_s, t_e)>, where K is a token that is a set of one or more RDF triples and t_s, $t_e \in T$ are the start and end timestamps forming a time interval. These tokens are used both for representing partial results that are being passed around the system as well as representing completed results that are omitted from the reasoner. Because each token consists of one or more triples, the token can also be considered a graph, but we denoted it as a token to avoid confusion with graphs that represent raw input data.

As each token can consist of one or more triples, we differentiate between two types of internal streams. The data element in the first represents singleton graphs containing only one triple, so is called an internal triple stream, denoted as S_{it}, while the second is called an internal graph stream, denoted as S_{ig}, where tokens represent graphs containing more than one triple. The time element in both types is the same, which includes two timestamps representing a time interval. We note that internal triple streams are a subset of internal graph streams, which means that every internal triple stream is also an internal graph stream.

An internal stream is ordered by the start timestamp of its elements. As in external streams, there could be multiple—but limited—tokens with the same start timestamp in an internal stream. Furthermore, there is no order among tokens with the same start timestamp.

An external stream of RDF graphs can be transformed into an equivalent stream of RDF triples by splitting each graph into its constituent triples. The same timestamp t is assigned as the start time t_s to all triples of this graph so that ordering is preserved internally. The end timestamp t_e is defined as infinity ∞.

4.2.2 Operators

All operators in our continuous reasoning framework work on a data-driven basis. Each operator takes one or multiple streams as input and produces one stream as output. The difference between operators is that some of them can be stateless, while others must be stateful to work correctly.

According to Andrade et al. (2014), a stream processing operator can be either stateful (maintains an internal state across tuples during processing), or stateless (processes tuples independently from each other). Stateless operators, including filter, map, graph-to-triple, and window operators, can process each received stream element they receive without storing or accessing any internal data structure created by processing earlier data (Andrade et al., 2014). If the operation results in an output element, it is immediately appended to the output stream. On the other hand, stateful operators, including join and aggregation operators, need to access and maintain internal states each time they receive an input. These internal states affect the results produced by the operator (Andrade et al., 2014). In the next subsections, we provide an operational description of these operators, presenting a streaming algorithm for each of them. We start with the operators that convert between the different types of streams defined in the previous section, followed by data processing operators.

We expect these operators to be composed in networks as follows: first, an external-to-internal operator converts an external stream to an equivalent internal graph stream. This is followed by a graph-to-triples operator which converts the internal graph stream to its equivalent internal triple stream. The internal triple stream is then passed to any number of filter operators. The resulting internal triple streams from filters are then joined into internal graph streams by join operators. Window operators can be placed before or after the filters, but necessarily before joins. The final data processing operator in a network should be the map operator which generates the results as an internal graph stream. This stream can then be passed to an internal-to-external operator to generate results in the form of an external stream, and can enter the network again for further processing as an internal graph stream through the graph-to-triples operator.

4.2.2.1 External-to-internal

Operator: External-to-internal

Input: stream S_e^{in}

Output: stream S_{ig}^{out}

1 Foreach $<G^{RDF}, t>$ arriving from S_e^{in}

2 Append $<G^{RDF},[t,\infty)>$ to S_{ig}^{out}

Listing 4.1: External-to-internal operator

This unary operator is responsible for transforming an external stream to its equivalent internal graph stream ready to be consumed by other operators. For each arriving graph, it assigns the timestamp of the graph as the start time and infinity as the end time forming the time interval $[t_s, t_e)$. The output stream elements of this operator (the internal graph stream) are ordered by the start timestamp.

Example. Listing 4.2 (a) represents an example external stream input, and Listing 4.2 (b) represents the output stream of an external-to-internal operator.

```
<{(s1,p1,o1), (s1,p2,o2)}, 5>
<{(s2,p1,o4), (s2,p2,o4), (s2, p3, o5)}, 8>
```

Listing 4.2(a): Example external stream input

```
<{(s1,p1,o1), (s1,p2,o2)}, [5, ∞)>
<{(s2,p1,o4), (s2,p2,o4), (s2, p3, o5)}, [8, ∞)>
```

Listing 4.2(b): Output stream of an external-to-internal operator

4.2.2.2 Graph-to-triples

Operator: Graph-to-triples

Input: stream S_{ig}^{in}

Output: stream S_{it}^{out}

1 Foreach $<G^{RDF}, [t_s, t_e)>$ arriving from S_{ig}^{in}

2 Foreach $(s,p,o) \in G^{RDF}$

3 Append $<\{(s,p,o)\},[t_s, t_e)>$ to S_{it}^{out}

Listing 4.3: Graph-to-triples operator

This unary operator is responsible for transforming an internal graph stream to its equivalent internal triple stream. It deconstructs the incoming RDF graphs into their individual triples and for each graph assigns the time interval of the graph as the time interval to each triple. The output stream elements of this operator are ordered by the start timestamp.

Example. Listing 4.4 (a) represents an example input stream, and Listing 4.4 (b) represents the output stream of a graph-to-triples operator.

```
<{(s1,p1,o1), (s1,p2,o2)}, [5, ∞)>
<{(s2,p1,o4), (s2,p2,o4), (s2,p3,o5)}, [8, ∞)>
```

Listing 4.4(a): Example input stream of a graph-to-triples operator

```
<{(s1,p1,o1)}, [5, ∞)>
<{(s1,p2,o2)}, [5, ∞)>
<{(s2,p1,o4)}, [8, ∞)>
<{(s2,p2,o4)}, [8, ∞)>
<{(s2,p3,o5)}, [8, ∞)>
```

Listing 4.4(b): Example output stream of a graph-to-triples operator

4.2.2.3 Internal-to-external

Operator: Internal-to-external

Input: stream S_{ig}^{in}

Output: stream S_{e}^{out}

1 Foreach $< G^{RDF},[t_s, t_e)>$ arriving from S_{ig}^{in}

2 Append $<G^{RDF},t_s>$ to S_{e}^{out}

Listing 4.5: Internal-to-external operator

This operator is responsible for transforming an internal graph stream back to its equivalent external stream. For each arriving graph, it removes the end timestamp in order to get a stream of graphs with single timestamps, matching definition 3 of an external stream.

Example. Listing 4.6 (a) represents an example internal stream input, and Listing 4.6 (b) represents the output stream of an internal-to-external operator.

```
<{(s1,p1,o1), (s1,p2,o2)}, [5, 15)>
<{(s2,p1,o4), (s2,p2,o4), (s2, p3, o5)}, [8, 18)>
```

Listing 4.6(a): Example internal stream input

```
<{(s1,p1,o1), (s1,p2,o2)}, 5>
<{(s2,p1,o4), (s2,p2,o4), (s2, p3, o5)}, 8>
```

Listing 4.6(b): Output stream of an internal-to-external operator

4.2.2.4 Filter

Operator: Filter
Input: stream S_{it}^{in}, filter predicate fp
Output: stream S_{it}^{out}
1 Foreach <{(s,p,o)},[t_s,t_e)> arriving from S_{it}^{in}
2 If fp(<{(s,p,o)},[t_s,t_e)>) is true
3 Append <{(s,p,o)},[t_s,t_e)> to S_{it}^{out}

Listing 4.7: Filter operator

The unary filter operator, in general, evaluates a predicate over each incoming element. If the predicate is satisfied, the element is immediately appended to the output stream; otherwise, it is simply discarded. The predicate is general enough to capture any one-pass, externally defined predicate (including the built-in datatype predicates and functions of RIF Core) and also the common case of triple patterns. When the predicate represents a triple pattern, the filter operator is semantically equivalent to SPARQL's basic pattern match.

Let V denote the set of variables, I the set of IRI constants, L the set of literals, B the set of blank nodes. Let TPN = {(s,p,o)| s ∈ I∪B∪V, p ∈ I∪V, o ∈ I∪L∪B∪V} be the set of triple patterns (Harris et al., 2013). Given tpn ∈ TPN, the triple pattern match predicate is evaluated as follows:

$$match((s,p,o,[ts,te)),tpn) = \begin{cases} true, & (s = tpn.s \vee tpn.s \in V) \wedge \\ & (p = tpn.p \vee tpn.p \in V) \wedge \\ & (o = tpn.o \vee tpn.o \in V) \\ false, & otherwise \end{cases}$$

It ensures that the coming triple's subject is equal to the triple pattern's (tpn) subject or that the triple pattern's subject is a variable (in the following examples, we use the question mark '?' to mark a variable name). The same checks are done for the predicate and object parts of the coming triple. The algorithm presented in Listing 4.7 describes the filtering process.

As the input stream is handled triple by triple in order, and as the time annotation is not changed, the same order is preserved in the output stream.

Example. Listing 4.8 (a) represents an example input stream, and Listing 4.8 (b) represents the output stream of a filter operator, with the triple pattern (?x, p2, ?y) as its filter predicate.

```
<{(s1,p1,o1)}, [5, ∞)>
<{(s1,p2,o2)}, [5, ∞)>
<{(s2,p1,o4)}, [8, ∞)>
<{(s2,p2,o4)}, [8, ∞)>
<{(s2, p3,o5)}, [8, ∞)>
```

Listing 4.8(a): Example input stream of a filter operator

```
<{(s1,p2,o2)}, [5, ∞)>
<{(s2,p2,o4)}, [8, ∞)>
```

Listing 4.8(b): Output stream of a filter operator (fp=(?x, p2, ?y))

4.2.2.5 Window

Operator: Window

Input: stream S_{it}^{in}, window size w

Output: stream S_{it}^{out}

1 Foreach $<K,[t_s,t_e)>$ arriving from S_{it}^{in}

2 Append $<K,[t_s,t_s+w)>$ to S_{it}^{out}

Listing 4.9: Window operator (time-based)

The window operator is widely used in stream processing for two main reasons. First, it unblocks stateful operators and constrains the unlimited memory requirement. All the above one-pass operators are easily adapted to process streams in a pipelined fashion, as they do not have to keep states. However, stateful operators need to access the whole state in order to produce results. If streams are infinite, the memory required to save the state is unlimited, and the operator will be blocked indefinitely as it waits for the end of the streams. The window operator restricts the size of the input stream. The second reason for using the window operator is that it can serve as a part of the query semantics, as streaming applications are usually concerned with pattern changes over time. Users, for example, can express that they are interested in a specific event happening in the last hour as part of the query or rule.

Listing 4.9 shows the window operator algorithm, where the time-based window size is denoted as w, which is a natural number ($w \in N$) that represents the number of time instances covered by the window. The window operator, itself, is a stateless one-pass unary

operator. We use a time-based sliding window definition. Its task is to assign the expiration time of each incoming element by adding the specified window size to the element's start time. The window operator can be placed anywhere in the query pipeline but should precede any stateful operator. These stateful operators use the expiration time assigned by the window operator to maintain their windowed states.

We note that this model does not place a limit on the number of stream elements that fit within a windowed state. Referring back to the external stream definition (Definitions 3), which states that multiple graphs can have the same timestamp, it means that there may be a very large but finite number of graphs within a windowed state at any given point. However, we do not consider windows that are fixed in terms of the number of stream elements they hold (such as the triple-based windows defined in CQELS (Le-Phuoc et al., 2011)).

Example. Listing 4.10 (a) represents an example input stream, and Listing 4.10 (b) represents the output stream of a window operator, with the size defined as 10 time instances.

```
<{(s1,p1,o1)}, [5, ∞)>
<{(s2,p1,o4)}, [8, ∞)>
<{(s3, p3,o5)}, [10, ∞)>
```

Listing 4.10(a): Example input stream of a window operator

```
<{(s1,p1,o1)}, [5, 15)>
<{(s2,p1,o4)}, [8, 18)>
<{(s3, p3,o5)}, [10, 20)>
```

Listing 4.10(b): Example output stream of a window operator (w=10)

4.2.2.6 Join

Join is a binary stateful operator. It works symmetrically by matching arriving elements from one input stream to elements of the state of the opposite stream. Unlike traditional joins, joins in streaming applications do not work on full states but rather on windows representing the most recent part of the input stream. As streaming data change over time, data elements continuously enter and exit the valid window part of the stream. Therefore, it is the join operator's responsibility to ensure that their window states are up to date. At any point in time $t \in T$, a window state of a stream i is $ws_i = \{<K, [t_s,t_e)> \mid <K, [t_s,t_e)> \in i \wedge t \leq t_e\}$, i.e. it includes all elements in stream i that have not expired. The join operator uses a join predicate, jp, which is a condition composed of variables and constants, conjunction and disjunction symbols ($\wedge$, $\vee$), in addition to equality and ordering symbols ($=$, $\neq$, $<$, $>$, $\leq$, $\geq$).

```
Function: removeExpired(window state ws, timestamp t)
1 ex = new list
2 Foreach <K,[ts,te)> ∈ ws
2       if(te < t)
3               add K to ex
4               remove <K, [ts,te)> from ws
5       else return ex
```

Listing 4.11: Auxiliary function 'removeExpired' algorithm

```
Function: probe(window state ws, element K, join predicate jp)
1 return all K' ∈ ws where jp(K, K') holds
```

Listing 4.12: Auxiliary function 'probe' algorithm

```
Operator: Join (left activation)
Input: streams S_igl^in, S_itr^in, window states ws_l, ws_r, join predicate jp
Output: stream S_ig^out
1 Foreach <K,[ts,te)> arriving from S_igl^in
2       insert <K,[ts,te)> to ws_l
3       removeExpired(ws_r, ts)
4       matches = probe(ws_r, K, jp)
5       foreach <K',[ts', te')> ∈ matches
6               Append <K∪K', [max(ts,ts'),min(te, te'))> to S_ig^out
```

Listing 4.13: Join operator (left activation)

There are four processing steps that are carried out symmetrically by a join operator upon each arrival of a stream element. First, the arriving element is added to the state of the side it has arrived from. Second, the opposite stream state is updated by removing expired elements. This is to ensure that the new element does not match an outdated element. Then, the opposite state is probed for matches. Finally, the results are generated and appended to the output stream. Listing 4.13 shows these steps when a stream element arrives at the left side. Line 2 inserts the new element to the left window state, lines 3 and 4 update and probe the right window state, and lines 5 and 6 generate results. There is an equivalent

right hand join algorithm to handle elements arriving at the right input stream. The full join requires both of these algorithms, so they both append results to the same output stream.

Multiple join operators can be connected to create a left-deep network. Therefore, apart from the first join in the network, all other joins receive an internal graph stream—which is the output of the previous join in the network—as their left input, and an internal triple stream—which is usually an output of a filter operator—as their right input. The first join in the network, as a special case, receives two internal triple streams from two filter operators. However, the join operator signature should still be adequate, as we defined internal triple streams as a subset of internal graph streams. As this first join constructs its results by combining an arriving triple from one side with a matching triple from the window state of the other side, the result is an internal graph stream. This means that the first join operator works also as a triple-to-graph operator.

Deciding which elements to remove from a window state is based on the expiration times attached to these elements, as shown in the 'removeExpired' function (Listing 4.11). Elements with expiration times that are smaller than the start times of the newly arriving tuples cannot be joined, as there is no overlap in their validities. As input streams arrive in increasing order of start time, even future elements will have no chance to join with them, so they can be safely removed.

When joining two stream elements, the result is a token that contains both elements. This generated element should be considered invalid as soon as one of the elements that contributed to it expires. Therefore, the generated token's time interval is assigned as the intersection of the intervals of the contributing elements.

The join algorithm is symmetrical; it handles elements from both sides in the order in which they arrived. However, it is possible for the join operator to receive an element from each side at the same time (thus having the same timestamp). We show here that the result stream will be the same regardless of the order in which the join operator handles its input. Let e_l be the new element arriving at the left input stream, e_r the new element arriving at the right input stream, and ws_l and ws_r the left and right window states. Now, let us consider joining the left input element first. The algorithm starts by inserting e_l in the left window state, ws_l:

1- $ws_l' = ws_l \cup e_l$

The next step is to update the right window by removing expired elements (referred to as ex):

2- $ws_r' = ws_r - ex_r$

Then, the updated right window is probed with the left incoming element to identify matches:

3- matches = $e_l \bowtie ws_r'$

At this stage, the algorithm finishes by appending the found matches to the result stream S1:

4- S1 = {earlier elements, $e_l \bowtie ws_r'$}

The algorithm is called again, repeating the steps performed on the other side for the element arriving at the right input, e_r. First, the updated right window state, ws_r' (step 2), is updated again by inserting the new element:

5- $ws_r'' = ws_r' \cup e_r$

Then, the left window (which was updated in step 1) is updated by removing expired elements:

6- $ws_l'' = ws_l' - ex_l$

New matches are found by probing the updated left window with the element arriving at the right input:

7- matches = $e_r \bowtie ws_l''$

The result stream, S1, is then appended with the new matches:

8- S1 = {earlier elements, $e_l \bowtie ws_r'$, $e_r \bowtie ws_l''$}

The second possibility is that the join algorithm starts processing the element from the right stream first, so let us repeat the same eight steps to produce output stream S2 so we can determine whether S1 and S2 are equivalent.

1- $ws_r' = ws_r \cup e_r$

2- $ws_l' = ws_l - ex_l$

3- matches = $e_r \bowtie ws_l'$

4- S2 = {earlier elements, $e_r \bowtie ws_l'$}

5- $ws_l'' = ws_l' \cup e_l$

6- $ws_r'' = ws_r' - ex_r$

7- matches = $e_l \bowtie ws_r''$

8- S2 = {earlier elements, $e_r \bowtie ws_l'$, $e_l \bowtie ws_r''$}

To compare S1 with S2, we disregard the earlier elements, as they are supposed to be the same. Thus, we compare S1 = { $e_l \bowtie ws_r'$, $e_r \bowtie ws_l''$} with S2 = { $e_r \bowtie ws_l'$, $e_l \bowtie ws_r''$}. From steps 1, 2, and 6 above, S1 = { $e_l \bowtie ws_r'$, $e_r \bowtie ws_l''$} = { $e_l \bowtie (ws_r - ex_r)$, $e_r \bowtie ((ws_l \cup e_l) - ex_l)$ } = { $e_l \bowtie (ws_r - ex_r)$, $(e_r \bowtie (ws_l - ex_l)) \cup (e_r \bowtie e_l)$ } and S2 = { $e_r \bowtie ws_l'$, $e_l \bowtie ws_r''$} = { $e_r \bowtie (ws_l - ex_l)$, $e_l \bowtie ((ws_r \cup e_r) - ex_r)$} = { $e_r \bowtie (ws_l - ex_l)$, $(e_l \bowtie (ws_r - ex_r)) \cup (e_l \bowtie e_r)$}. If we disregard the part that joins e_l with e_r as it is the same in both streams (as the join operator is commutative) we find that S1 = { $e_l \bowtie (ws_r - ex_r)$, $(e_r \bowtie (ws_l - ex_l))$} and S2 = { $e_r \bowtie (ws_l - ex_l)$, $(e_l \bowtie (ws_r - ex_r))$}.

We note that both streams contain the same results, but appear in a different order. Based on the definition of the join operator, the join results are stamped with the largest start timestamp of its parents. As we expect stream elements to arrive in order, e_l will have a larger start timestamp than those in the right window state. Therefore, elements in $(e_l \bowtie (ws_r - ex_r))$ should use e_l's start timestamp as their start timestamp. The same is true on the other side; elements in $(e_r \bowtie (ws_l - ex_l))$ should use e_r's start timestamp as their start timestamp. As e_l and e_r have the same start timestamp, both streams are considered equivalent. Definition 4 states that there is no order among stream elements with the same start timestamp.

Example. Using a 10 time instances window, consider this simple join example: (?x, p2, ?y)∧(?x, p2, ?z). It joins two triples (with p2 as a predicate) if they have the same subject. Listing 4.14 (a) and (b) represents left and right input streams of a join operator, and Listing 4.14 (c) represents its output stream.

```
<{(s1,p2,o1)}, [5, 15)>
<{(s2,p2,o4)}, [8, 18)>
<{(s4, p2,o5)}, [11, 21)>
<{(s3, p2,o5)}, [14, 24)>
<{(s2, p2,o7)}, [17, 27)>
```

Listing 4.14(a): Example left input stream of a join operator

```
<{(s2,p2,o1)}, [6, 16)>
<{(s3,p2,o2)}, [10, 20)>
<{(s1,p2,o3)}, [12, 22)>
<{(s4,p2,o2)}, [14, 24)>
<{(s5,p2,o4)}, [15, 25)>
```

Listing 4.14(b): Example right input stream of a join operator

```
<{(s2,p2,o4), (s2,p2,o1)}, [8, 16)>
<{(s1,p2,o3), (s1,p2,o1)} [12, 15)>
<{(s3, p2,o5), (s3,p2,o2)}, [14, 20)>
<{(s4,p2,o2), (s4, p2,o5)}, [14, 21)>
```

Listing 4.14(c): Example output stream of a join operator

When the second element from the left stream (with subject s2) arrives, it generates the first result as it joins with the first element in the right window. The second result is generated when the third element from the right stream (with subject s1) joins the first element in the left window state. Then at time point 14, we simultaneously get two elements from both streams. If we handle the incoming element from the left stream first,

the new element with subject s3 joins with the second element in the right window state, creating the result <{(s3, p2,o5), (s3,p2,o2)}, [14, 20)>, then by handling the incoming element from the right stream, we get the result <{(s4,p2,o2), (s4, p2,o5)}, [14, 21)>. On the other hand, if we handle the incoming element from the right stream first, the result of s4 subject with the time interval [14, 21) will be generated first, followed by the result of s3 subject with the time interval [14, 20). As they both have the same start timestamp (which defines the order of the stream), we will get the same output stream regardless of which input side we handle first. Finally, notice that the last incoming element from the left stream does not join with the first element in the right side as it expires before the arrival of the new element.

4.2.2.7 Aggregation

Operator: Aggregation

Input: stream S_{ig}^{in}, window state ws, current aggregate v, aggregation function agf

Output: stream S_{ig}^{out}

1 Foreach <K,[t_s,t_e)> arriving from S_{ig}^{in}

2 removeExpired(ws, t_s)

3 t_e' = smallest end time in ws

4 insert < K,[t_s,t_e)> to ws

5 v = agf(ws)

6 append <K∪{(currentAggregate, hasValue, v)}, [t_s, min(t_e, t_e'))> to S_{ig}^{out}

Listing 4.15: Aggregation operator

Aggregate operators provide and maintain statistics about window states. They are widely used in streaming applications to provide compact summaries of the streams. These statistics include count, sum, max, min, and average (which can be derived from sum and count). The operator uses a user-defined aggregate function (agf) that is applied successively to the current window state, ws. In the most general algorithm (presented in Listing 4.15), the aggregate operator needs to maintain one window (the current window state) and keep all its elements in order to re-evaluate the aggregate function with the updated content of the window. Similar to the join operator, upon each arrival of a new element, an aggregate operator needs to first update its window state by removing expired elements. Then, it inserts the arriving element to its window state, re-evaluates its function on the current window, and constructs a new triple around the new result to be appended,

along with the arriving token, to the output stream. An example aggregate function agf is max (presented in Listing 4.16), which finds the maximum value of a specific variable in the current window. We use the notation K(x) to refer to the attribute of interest 'x' in the arriving token 'K'. For example, if K is {(s1,p1,o1), (s2,p2,20)}, and x is the object of the second triple in K, then K(x) = 20.

```
Function: max(window state ws)
1 // x is the attribute of K that contains the numerical value of interest
2 v = 0
3 Foreach <K,[ts,te)> ∈ ws
4       if(K(x) > v)
5               v = K(x)
6 return v
```

Listing 4.16: max aggregate function

While some aggregate functions can be maintained incrementally, some require a new re-evaluation by scanning the whole window in special cases. Sum and count can be maintained incrementally as follows: for each insertion, the new element's value is added to the current sum, and the count is incremented by one; for each deletion of an expired element, its value is decreased from the current sum, and the count is decremented by one (sum incremental algorithm is presented in Listing 4.17). In this case, we send the new element, the expired elements, and the current aggregate to the aggregate function instead of sending the whole window state. Therefore the aggregation algorithm in Listing 4.13 needs slight changes in line 2 where there should be a list (ex) to receive the expired elements from 'removeExpired' function (ex = removeExpired(ws, t_s)), and line 5 to send the new element K, the expired elements ex, and the current aggregate v instead of sending ws to the aggregate function (v = agf(K, ex, v)). Max and min, on the other hand, require rescanning the window upon each expiration, where the expired element is the current max or min.

Function: sum(new element K, expired elements ex, current aggregate v)

1 // x is the attribute of K that contains the numerical value of interest

2 Foreach <K',[t_s,t_e)> ∈ ex

3 v = v – K'(x)

4 v = v + K(x)

5 return v

Listing 4.17: sum aggregate function (incremental)

Example. Consider a simple aggregate function that finds the element with the maximum object value (e.g. a temperature or a vehicle speed) in a 10 time instances window. Listing 4.18 (a) represents an example input stream, and Listing 4.18 (b) represents the output stream. Notice that we receive the last element, the previous maximum value of 30 is already expired, so the algorithm finds the maximum value in the valid window state.

<{(s1,p1,20)}, [5, 15)>

<{(s3,p1,30)}, [10, 20)>

<{(s4,p1,20)}, [16, 26)>

<{(s5,p1,17)}, [21, 31)>

Listing 4.18(a): Example input stream of a 'max' aggregation operator

<{(s1,p1,20)}, [5, 15)>

<{(s3,p1,30)}, [10, 15)>

<{(s3,p1,30)}, [16, 20)>

<{(s4,p1,20)}, [21, 26)>

Listing 4.18(b): Example output stream of a 'max' aggregation operator

4.2.2.8 Map

Operator: Map

Input: stream S_{ig}^{in}, mapping function mf

Output: stream S_{ig}^{out}

1 Foreach <K,[t_s,t_e)> arriving from S_{ig}^{in}

2 Append <mf(K),[t_s,t_e)> to S_{ig}^{out}

Listing 4.19: Map operator

This operator applies a mapping function to each arriving token and appends the result to the output stream. The map function, mf, applies a user-defined graph template composed of elements from the set (I ∪ B ∪ L ∪ V) to transform an incoming token to a new graph.

It is a more general form of the relational algebra projection operator, as it can generate new attributes that are not part of the incoming element. In our rule-based reasoning context, this operator is used to generate the head of the rule when the token satisfies the whole body of this rule. This way, it is also equivalent to SPARQL's CONSTRUCT clause. The generated result keeps the same time interval of the original token.

Note that Definition 4 states that tokens can represent complete results. Therefore, the result stream produced by the map operator is represented as an internal stream so that it can re-enter the system while holding time intervals that model its validity.

Example. Listing 4.20 (a) represents an example input stream, and Listing 4.20 (b) represents the output stream of a map operator, that generates the head (?x, ?p, ?z) from the input {(?x, ?p, ?y),(?y, ?p, ?z)} which can be used to imply transitivity.

```
<{(s1,p1,o1), (o1,p1,o2)}, [5, 10)>
<{(s2,p2,o4), (o4,p2,o3)}, [8, 12)>
```

Listing 4.20(a): Example input stream of a map operator

```
<{(s1,p1,o2)}, [5, 10)>
<{(s2,p2,o3)}, [8, 12)>
```

Listing 4.20(b): Example output stream of a map operator

The time interval and expiration model are based on the temporal operator algebra of Krämer and Seeger (2005). However, their stateful operators' algorithms do not release output results until they expire to ensure correctness of results, as no future input elements can modify the output. As this does not follow the real-time requirement of our framework, our operators release results as soon as they are produced. This does not affect the correctness of the results, as it currently does not include operators that can invalidate previous results before they expire such as the difference operator.

4.3 R4: Rule-based Reasoner for RDF streams using Rete

We implemented the above framework of continuous reasoning for RDF streams using the Rete algorithm. The Rete algorithm is used for pattern matching in the rule-based reasoning process. Rules are translated into Rete networks of nodes. The nodes represent different operators that can be shared between rules and the data flows between these nodes. The tree-like network divides the matching process into multiple steps that perform different checks, so if a data element does not match the first node, it is simply discarded and does not complete its way through the network.

In the first stage, a discrimination network partitions the input streams by applying filtering conditions (filter nodes), forming Rete's alpha network. These streams are then fed into the beta network where multiple join nodes group their input streams that share the same value of a specified variable. Finally, after all the streams are joined, a terminal node is responsible for generating the new inferred statements, which are modelled to form the output stream. This stream then enters the network again as input in order to infer more results. The same process is applied for user defined rules and background ontological rules such as RDFS (Brickley et al., 2014) entailment rules. Figure 4.1 illustrates the Rete network operators and data flow to match rules 9 and 10 of the RDFS entailment rules.

The original Rete algorithm was not designed to deal with streaming data. In fact, Rete trades memory space for faster processing, as it materialises all intermediate results. In streaming applications, this is not a viable option, as streams can be of unlimited size. Therefore, we extend Rete by applying the concept of sliding windows to the working memories in a way equivalent to Berstel (2002). Each join node maintains a sliding window over each stream input, effectively replacing alpha and beta memories with stream window states.

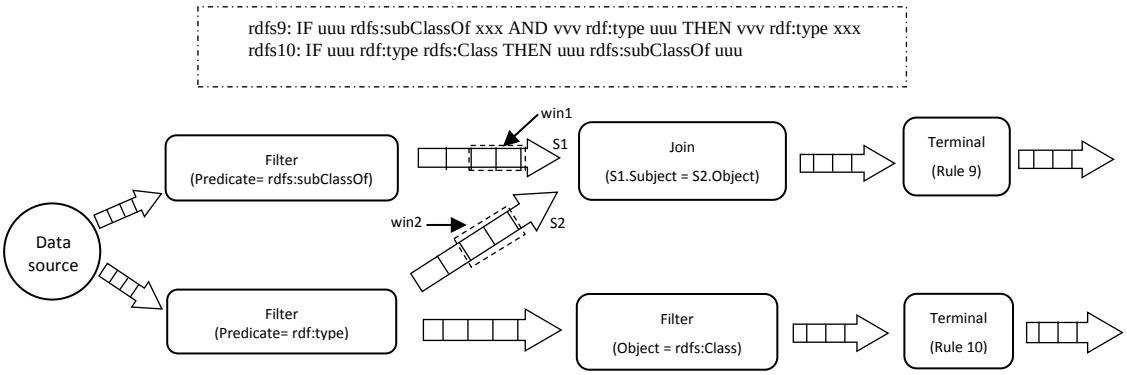


Figure 4.1: Rete network example (RDFS rules 9 and 11)

The next subsections present the rule language used to declare rules, give the system architecture, and detail the reasoning process carried out by the Rete networks.

4.3.1 Rule Language

For generic rule-based reasoning, we chose RIF (Kifer and Boley, 2013) to express the rules that are going to be continuously matched against data streams. Besides being a W3C standard that fits nicely with other Semantic Web standards (its compatibility with RDF and OWL is defined in de Bruijn and Welty (2013)), RIF was mainly designed to facilitate

rule exchange among other rule systems; however, we only consider it here as a declarative rule language. R4 supports RIF Core, which corresponds to the language of definite Horn rules without function symbols, equivalent to Datalog (Boley et al., 2013).

As data streams can be of unlimited size, rules cannot be evaluated against whole streams. A common approach to address this problem that our continuous reasoning framework follows is to specify a limited subpart – a window – of the stream. These windows convert an unlimited stream into a finite set, which a rule can be matched against. As these windows can form a part of the rule semantics, users should be able to specify them using the rule language. Therefore, we extend RIF to express these windows in the rule documents in order to be applied in streaming contexts.

Listing 4.21 shows the EBNF grammar for the extended RIF Core syntax. RIF rule sets are organised into Documents, Groups, and Rules. At the document level, the 'Import' directive is used to import data from non-RIF Core documents, such as RDF data or OWL ontologies, where the LOCATOR is a URI indicating the location of the imported document and PROFILE is an optional entailment regime. The import directive can be used to import the static background data that is temporally agnostic. To enable the system to differentiate between this data and streaming temporal data, we add the StreamImport directive, which defines streaming data sources. For each stream, users can specify the window size, i.e. the time validity for each element of this stream. For example, users can state that they are only interested in tweets posted during the last hour by a specific user. The window specification is optional, as some rules that do not block operators can be processed without time restrictions, e.g. simple filtering.

Specifying the window size at the import directive is generally semantically equivalent to specifying it at the FROM clause in SPARQL extensions such as C-SPARQL. They both extract from the stream the most recent elements that occur during the last number of time units, specified as the size of the window. However, in SPARQL extensions, this is specified for each query, while the stream import directive is defined at the document level, which is expected to contain more than one rule. Therefore, the import window serves as a global window across all rules. As different rules can require different window constraints, local window size can also be optionally set at the formula level to enable more flexibility. This is comparable to a Streaming SPARQL window at the query's GroupGraphPattern level. Local window values (if specified) override global window values in the respective rule.

Rule Language:

```
  Document        ::= IRIMETA? 'Document' '(' Base? Prefix* Import* ImportStream*
Group?')'
  Base            ::= 'Base' '(' ANGLEBRACKIRI ')'
  Prefix          ::= 'Prefix' '(' Name ANGLEBRACKIRI ')'
  Import          ::= IRIMETA? 'Import' '(' LOCATOR PROFILE? ')'
  ImportStream    ::= IRIMETA? 'ImportStream' '(' LOCATOR PROFILE? ')' Window?
  Group           ::= IRIMETA? 'Group' Strategy? Priority? '(' (RULE | Group)* ')'
  RULE            ::= (IRIMETA? 'Forall' Var+ '(' CLAUSE ')') | CLAUSE
  CLAUSE          ::= Implies | ATOMIC
  Implies         ::= IRIMETA? (ATOMIC | 'And' '(' ATOMIC* ')') ':-' FORMULA
  LOCATOR         ::= ANGLEBRACKIRI
  PROFILE         ::= ANGLEBRACKIRI
```

Condition Language:

```
  FORMULA         ::= (IRIMETA? 'And' '(' FORMULA* ')' |
                       IRIMETA? 'Or' '(' FORMULA* ')' |
                       IRIMETA? 'Exists' Var+ '(' FORMULA ')' |
                       ATOMIC |
                       IRIMETA? Equal |
                       IRIMETA? Member |
                       IRIMETA? 'External' '(' Atom ')') Window? ('On' IRIMETA)?
  ATOMIC          ::= IRIMETA? (Atom | Frame)
  Atom            ::= UNITERM
  UNITERM         ::= Const '(' (TERM* ')'
  GROUNDUNITERM   ::= Const '(' GROUNDTERM* ')'
  Equal           ::= TERM '=' TERM
  Member          ::= TERM '#' TERM
  Frame           ::= TERM '[' (TERM '->' TERM)* ']'
  TERM            ::= IRIMETA? (Const | Var | List | 'External' '(' Expr ')')
  GROUNDTERM      ::= IRIMETA? (Const | List | 'External' '(' GROUNDUNITERM ')')
  Expr            ::= UNITERM
  List            ::= 'List' '(' GROUNDTERM* ')'
  Const           ::= '"' UNICODESTRING '"^^' SYMSPACE | CONSTSHORT
  Var             ::= '?' Name
  Name            ::= NCName
  SYMSPACE        ::= ANGLEBRACKIRI | CURIE
  Window          ::= positiveInteger TimeUnit
  TimeUnit        ::= 'ms' | 's' | 'm' | 'h'
```

Annotations:

```
  IRIMETA         ::= '(*' IRICONST? (Frame | 'And' '(' Frame* ')')? '*)'
```

Listing 4.21: Extended RIF Core syntax (extensions are shown in bold)

We also added another optional construct at the formula level. The ‘On’ construct specifies the ID of an input for this formula to be applied on. Using this option, different parts of the rules can be applied to different input streams, without having to apply all rules to all input streams.

4.3.2 System architecture

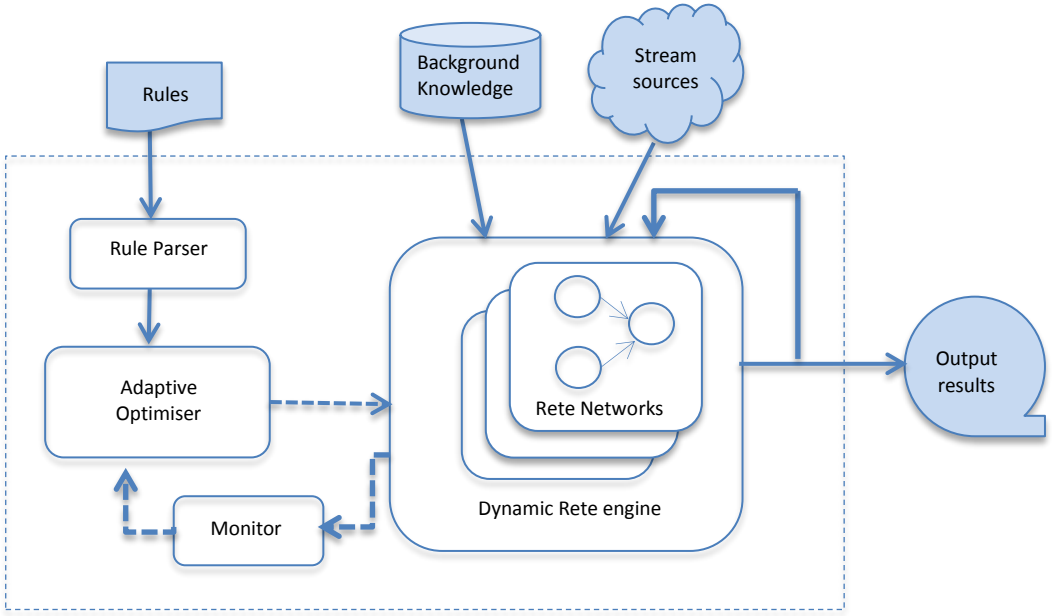


Figure 4.2: R4 system architecture

R4 is a native rule-based reasoner for RDF streams. It receives rules from users, evaluates them on RDF streams, and continuously provides results. The white-box system architecture is illustrated in Figure 4.2. The heart of the system is the dynamic Rete engine, where the continuous processing of both RDF streams and RDF static data actually happens. The engine instantiates the data flow network operators and connects them as specified by the optimiser. The network operators start processing the data pushed to the engine by the data sources. The engine keeps processing incoming data, generating streams of results, until it is explicitly requested to stop. While the generated results are pushed instantly as output to the user, they are also fed back to the network to be used as streaming input.

The second main component of R4 is the adaptive optimiser. Its main job is to choose an efficient plan for the current conditions to be used by the Rete engine. Apart from communicating the chosen plan to the rule engine, the optimiser communicates with two more components: the rule parser and the monitor. A rule document containing any number of rules in the extended RIF Core language is submitted to the system. The rule parser component translates the rules into abstract syntax trees and passes it to the optimiser. Then, the optimiser uses some basic heuristics to generate an initial plan and conveys it to the rule engine. As this initial plan might not be optimal, and as stream characteristics (e.g. input rates) can change over time, the monitor continuously collects

performance-related statistics about the running operators and sends them to the optimiser periodically. The optimiser uses the collected statistics to check if any changes need to be made over the running plan and instructs the Rete engine to perform such changes.

The following subsection describes the rule engine implementation of the continuous reasoning framework in more details, while the adaptive optimisation part is left for the next chapter.

4.3.3 Data Processing using Rete

The Rete-based dataflow networks of R4 consist of nodes that fall into four main categories: source nodes, alpha nodes, beta nodes, and terminal nodes. Each type carries out specific tasks, effectively implementing the operators detailed in Section 4.2.2.

Reading input: In R4, data enters the network through the source nodes. For each import in the RIF file, the rule engine instantiates a source node that is responsible for acquiring the data input from the imported document. As inputs can be static RDF data or streaming data, source nodes have two types: static sources and streaming sources. For static imports, source nodes pull the data (RDF statements) from the specified documents and push them to the nodes in their respective network. Therefore, static data populate the network before dealing with streaming data. For streaming imports, source nodes implement the graph-to-triples operator, converting streams of graphs into streams of triples. Furthermore, if the streaming import statement declares a window size (global window) for this stream, the corresponding source node also implements the window operator, annotating the incoming statements with time intervals. The start time is given as the system time of arrival if the parent graph was not already annotated at its origin, otherwise passing the annotation of the graph to the resulting triples. In both cases, the end time is calculated as the sum of the start time and window size. Source nodes propagate the annotated triples to their successor nodes, which are the alpha nodes.

Alpha network: Alpha nodes are single-entry nodes that form a discrimination network. In R4, an alpha node can receive RDF triples (streams or static) from any number of nodes through its single input. Each received element is matched against some conditions and is either dropped if there is no match or propagated downstream to its successor node(s) in the event of a match. For triple pattern conditions that contain, for example, two constants (e.g. (?x, p, o)), it is possible to either create two successive alpha nodes – one for each constant – or create one alpha node that checks both constraints. While the first option

increases the possibilities of sharing, we opted for the second option to get fewer nodes and, therefore, less traffic. The optimiser creates one alpha node for each triple pattern in the set of rules (triple patterns are represented as frames in RIF). Alpha nodes, therefore, implement the filter operator defined in 4.2.2.2. Alpha nodes treat static RDF triples and RDF streams equally, as they are not concerned with the time element. Each alpha node is followed by an alpha memory. The first alpha memory in the network is followed by a special node called the left input adaptor (described below), while the other alpha memories are followed by beta nodes.

Beta network: The left input adaptor node is responsible for preparing the incoming triples to enter the beta network from the left input of the first beta node in a given join-tree. It creates a new token that represents a partial match – defined as a list of triples – for each rule body, for which the incoming triple is a sub-graph match, then adds this triple as the first item in each list. It also annotates each token with the same time interval as the triple for which it was generated. Tokens are then sent to the first node in the beta network.

Beta nodes are two-input nodes that are responsible for joining the branches of the alpha network. In our implementation, as in the original Rete algorithm, beta nodes form a left-deep tree. Each beta node maintains a left memory, which is a beta memory storing tokens received from other beta nodes (or from the left input adaptor node in case of the first beta node), and a right memory, which is an alpha memory storing triples received from alpha nodes. Join nodes are beta nodes that implement the join operator (defined in 4.2.2.5). They match inputs from both sides according to some conditions, e.g. a shared variable binding. As explained earlier, we use window-join operators to avoid storing and operating over all partial results. In this context, each left (beta) and right (alpha) memory is implemented as a valid window state.

When a join node is left-activated (i.e. it receives a token through its left input), it first adds the new token to its left window then prepares its right window by removing any expired items (items with an end time that is earlier than the start time of the incoming token) so that no outdated results are matched. The right window is then searched according to the join node conditions. When a match is found, a new token is created by duplicating the left token and adding the right triple to the new token's triple list. The new token is annotated with the latest start time and earliest end time of the left token and right triple time annotations (the intersection of their time intervals) and then propagated to the next beta node, or to the terminal node if it is in the root node of a join tree. Conversely, when the join node is right-activated by receiving a triple from an alpha node, it adds the new triple

to the right window, prepares the left window, then matches the triple against the remaining tokens. Join nodes that join a streaming input with a static input can only be activated from the streaming side, and the join result is annotated with the same time interval of the arriving stream element.

Generating results: Finally, terminal nodes receive tokens from the root nodes of join-trees and are responsible for producing entailed graphs. The optimiser creates one terminal node for each rule, so a terminal node is an implementation of the map operator (defined in 4.2.2.3) using the rule head as the map function. The annotation of the entailed graphs is directly inherited from the completed token from which it is produced. The union of the output of all terminal nodes, which is the output of the system, itself, is re-entered as an input stream to the source nodes to support iterative inference.

Memories implementation: The data structures used to implement the window states need to support fast update (insertion and removal) and efficient search. Implementing the window states as priority queues, in which tokens or triples are ordered according to their end time annotations, ensures the efficient removal of expired elements, as the algorithm does not have to traverse the whole data structure searching for expired elements. However, this can perform badly with regards to probing, as the size of these memories is expected to be big and the match operation is repeated excessively with fast input rates. Therefore, we implemented each window state using a priority queue and a hash map in order to simultaneously improve the speed of updating (using the priority queue) and probing (using the hash map). Static memories, however, only need hash maps.

Each memory state supports three operations: insert, prune, and probe. Insert and prune work on both structures, while probe only uses the hash map. Listing 4.22 presents algorithms of these operations for alpha memories. If the rule's document only specifies a global window, then the insert operation simply inserts the arriving triple to the hash map and priority queue. However, if the alpha memory corresponds to a triple pattern (of which the enclosing formula contains a local window), the insert operation overrides the end time of the triple to represent the local window size.

Structure: Alpha memory

H is the memory's hash map; PQ is the priority queue; w is the local window size

Procedure: Insert (Triple e)

1 if e is associated with a timestamp

2 add e to PQ

3 if w ≠ null

4 $e.t_e = e.t_s+w$

5 add e to H

Procedure: Probe (Token k)

1 return H.get(k.joinAttribute)

Procedure: Prune (int t_s)

1 e = head of PQ

2 while $e.t_e<t_s$

3 remove e from PQ and from H

4 e = next triple in PQ

Listing 4.22: Alpha memory operations

Sharing memories: The Rete algorithm allows sharing nodes and their memories between different rules to save processing and memory costs. Nodes that have the same conditions and ancestors can be shared instead of duplicating. For example, if the triple pattern (?x, p, ?y), where p is a specific property, appears in two rules, the same alpha node that checks for this pattern and the resulting alpha memory can be shared between the two rules. However, a situation to be considered is if one of the identical triple patterns appears in a formula that specifies a local window, while the other one does not (i.e. it uses the global window), or if both of them appear in formulas with local windows of different sizes. In this case, the alpha memory adopts the bigger window so that no possible results are missed. As this can mean generating false positives for the formula with the smaller window size, the following join node corresponding to the smaller window formula needs to check for an overlap in the time intervals of its two inputs according to its own window definition during the probing process.

While the model allows different sized local windows to appear in the same rule, we notice that the join order can affect the number of results. To illustrate with an example, consider the following rule that has two joins, each specifies its own local window size: join events

a and b (a⋈ b) occurring in a 5 minute window, and join events b and c (b⋈ c) occurring in a 10 minute window.

First, elements from stream a will pass a window operator, which assigns an end timestamp by adding 5 to the start time (assuming the time unit is minutes). As the alpha memory of stream b is shared between two windows, it will take the bigger one, adding 10 to the start time. Elements from stream c will be assigned end timestamps by adding 10 to the start timestamp.

Consider these inputs: a= <a1,1>, b= <b1,2>, c= <c1,5>, <c2,8>. We should expect to get two answers from these inputs, the first one combines a1, b1, and c1, and the second one combines a1, b1, and c2.

By applying the window operators to the inputs, we get: a= <a1, [1,6)>, b= <b1,[2, 12)>, c= <c1,[5,15)>, <c2,[8,18)>. To join all streams in a left deep plan, we have two options: (a⋈b) ⋈c, or (b⋈ c) ⋈a.

In the first case: the first join (a⋈ b) generates a single result: <(a1,b1),[2,6)> and passes it to the second join, then the second join ((a⋈ b) ⋈c) re-assigns the expiration time of this element converting it to <(a1,b1),[2,12)> and matches it against the event stream c generating the results: <(a1,b1,c1),[5,12)> and <(a1,b1,c2),[8,12)>.

In the second case: the first join (b⋈ c) generates two results: <(b1,c1),[5,12)> and <(b1,c2),[8,12)> and passes them to the second join, then the second join ((b⋈ c) ⋈a) re-assigns the expiration time of these elements converting them to <(b1,c1),[5,10)> and <(b1,c2),[8,13)> and matches them against the event stream a generating the result: <(a1,b1,c1),[5,6)>. We notice that this join order misses the second result that combines a1, b1 and c2. While the first plan was able to generate this result by re-assigning the expiration time according to its local window, this plan still missed this output. Therefore, we note that when windows are of the same size, then the order of joins does not affect the result. On the other hand, when windows are of different sizes, the order of joins can affect the result, and hence reasoning may be incomplete.

Garbage collection: The join node algorithm ensures that data elements in one of the input memories that cannot be used to produce results are deleted by checking the expiration time every time a new element arrives at the other input. This works efficiently for join nodes that join two streams with similar input rates. However, if one of the input streams is very slow and the second is fast, the fast stream's memory can grow big before an element

arrives from the slow stream to trigger the garbage collection process. While this has no functional effect on the behaviour of the join, it does increase the resource consumption of this join. In this case, the join node should periodically check its input stream memories for expired data. Another extreme case is when a join node joins a static memory with a stream; this join is only activated from the streaming side, so garbage collection on the streaming memory is never triggered. For these joins, instead of invalidating expired elements from the opposite input memory, this is performed on the input memory of the same side, as the opposite memory is static and its elements are persistent. We note that this technique is not needed in the general case (where both streams have comparable arrival rates), as pruning the other input (which is always required in order to ensure correctness of results) is sufficient.

Ontology reasoning: In the RIF import statements, users can choose the entailment regime for background reasoning. If one is specified, the system builds a separate Rete network that implements the entailment rules of the specified regime. These networks feed their output entailments directly to the main Rete network that implements the user-defined rules. RDFS ontologies are supported by building a network that implements the 13 RDFS entailment rules. Appendix A shows the RDFS++ rules written in RIF and the corresponding Rete. RDFS++ rules go beyond basic RDFS constructs by supporting OWL inverseOf, sameAs, and TransitiveProperty (Allemang and Hendler, 2008). To support OWL ontologies, the RIF document 'OWL 2 RL in RIF' (Reynolds, 2013) describes two approaches to reason over the rule-based dialect of OWL 2 using RIF. The first is a direct translation of OWL 2 RL rules to RIF Core. While this approach is straightforward and easy to implement, it has the disadvantage of creating a big network for the large list of rules; however, some parts of the network might never be used (the corresponding rule's constructs are never used in the ontology). The second approach avoids this problem by translating the source OWL 2 RL ontology to a specialised RIF Core rule set. An algorithm that handles the instantiation of the RIF rule set for a particular ontology is described in the 'OWL 2 RL in RIF' document. As this algorithm only uses TBox axioms (static knowledge), it can be implemented as it is in our stream reasoner.

4.4 Conclusion

This chapter has presented our main contribution: a continuous reasoning approach, and the implemented stream reasoning engine R4: a rule-based reasoner for RDF streams using Rete. R4 provides continuous inferencing capabilities natively over RDF streams for

generic rules expressed in our extension of RIF, which represents a minor contribution of this thesis. We used Java as a programming language to implement a prototype system of R4. The implemented prototype can successfully read RDF streams, use Rete networks to reason over them, and continuously produce results. In the next chapter, we provide a concrete scenario with a number of use cases, along with a comparative evaluation of the implemented reasoner.

Chapter 5: Evaluating R4

Chapter 4 described R4, a continuous rule-based reasoner for semantic streams. This chapter tests and evaluates the performance of the implemented prototype. Section 5.1 presents an evaluation scenario – describing the input datasets used in the evaluation process, along with a number of rules with different complexities – to test the system functionality. Then, Section 5.2 presents a comparative evaluation; we conduct several experiments to evaluate the system performance by comparing it to other reasoners. Firstly, in Section 5.2.1, we compare R4 performance to a static reasoner to show the advantage of the continuous reasoning approach. Then, in Section 5.2.2, we compare R4 to other state-of-the-art systems that are designed to reason over semantic streams.

5.1 Evaluation scenario

We obtained semantic streams from the SemsorGrid4Env[11] project. The main objective of this project is to design, implement, and deploy a service-oriented architecture and middleware that allows application developers to build open, large-scale, semantic-based sensor network applications for environmental management (Gray et al., 2009). It employs Semantic Web techniques to real-world, real-time data coming from heterogeneous sensor networks so that developers can use these sensors for other environmental management purposes than those they were originally expected to have.

5.1.1 Datasets

Meteorological and oceanographic data generated by the Channel Coast Observatory[12] (CCO) sensor network are made available in RDF format by SemsorGrid4Env's CCO API (Frazer et al., 2011). Every half hour, the CCO API publishes new semantic sensor observations obtained from 24 sensors deployed around the English Channel. These observations observe several properties, e.g. wave height, wave direction, and wind speed. The API makes these available as a collection of observations (i.e. sets of observations sharing some characteristics) (Garcia-Castro et al., 2012) for a given half-hour period. A published collection could comprise only observations, but could also be a collection that

[11] http://www.semsorgrid4env.eu/
[12] http://www.channelcoast.org/

aggregates other collections of observations. This data is published on the Web as linked data[13] and stored in a triple store with a SPARQL endpoint[14].

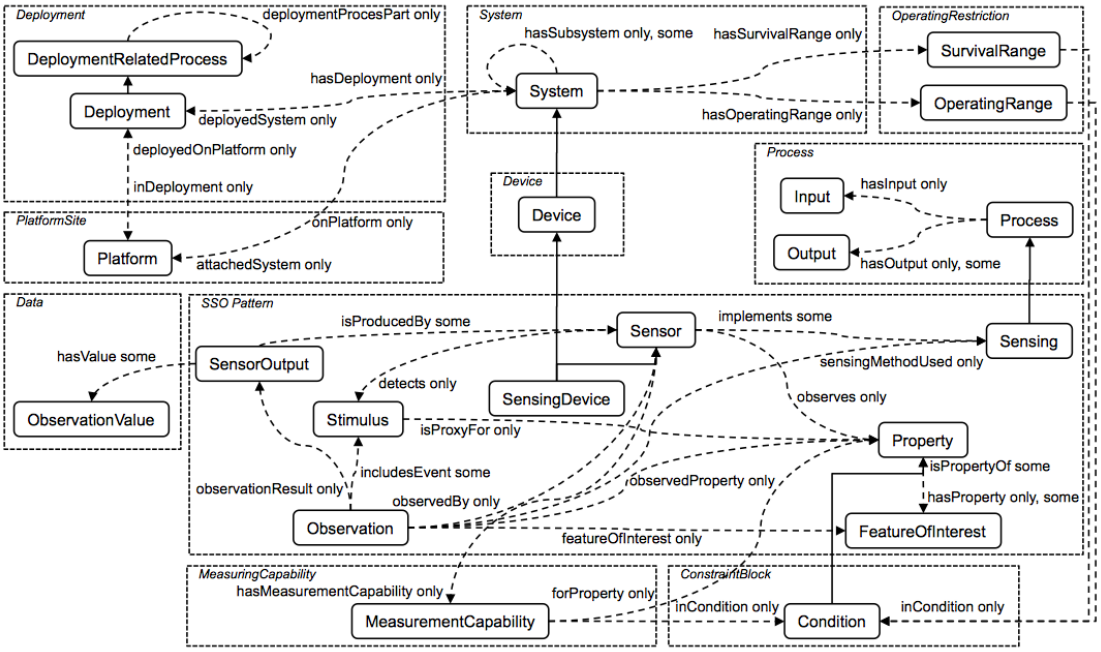


Figure 5.1: The SSN ontology, key concepts and relations, split by conceptual modules (Compton et al., 2012)

The project reuses a number of ontologies to model the data. However, it mainly relies on the Semantic Sensor Networks (SSN) ontology (Compton et al., 2012). The SSN ontology is organised into ten conceptual modules of related concepts, as shown in Figure 5.1. The core module is the Sensor-Stimulus-Observation (SSO) pattern, which links sensors, what they sense, and the resulting observations. Other modules are used to, for example, represent the deployment of sensors and their platforms, the measuring capabilities of sensors and the survival conditions of specific environments. The SSN ontology uses the DOLCE+DnS Ultra Lite (DUL)[15] foundational ontology as an upper ontology to facilitate interoperability. The SemSorGrid4Env project also uses an extension of the SSN ontology to model some aspects that are not covered by the SSN ontology, such as observation collections and measurement properties (Garcia-Castro et al., 2012), the SWEET ontology (Raskin and Pan, 2005) to describe the services provided by the infrastructure, and the Coastal Defence ontology (Garcia-Castro et al., 2012), which represents features of interest and their properties that are specific to the flood emergency planning use case.

[13] e.g., http://rdf.api.channelcoast.org/observations/cco/boscombe/Dirp/latest contains the latest wave direction observations by the Boscombe sensor.
[14] Available at http://env.ecs.soton.ac.uk:8000/
[15] http://www.ontologydesignpatterns.org/ont/dul/DUL.owl

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix w3time: <http://www.w3.org/2006/time#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix ssn: <http://purl.oclc.org/NET/ssnx/ssn#> .
@prefix ssnExt: <http://www.semsorgrid4env.eu/ontologies/SsnExtension.owl#> .
@prefix coastD: <http://www.semsorgrid4env.eu/ontologies/CoastalDefences.owl#> .
@prefix waves: <http://marinemetadata.org/2005/08/ndbc_waves#> .

<http://id.api.channelcoast.org/observations/cco/penzance/Dirp/20130326#000000>
  rdf:type ssn:Observation ;
  ssn:observedBy <http://id.api.channelcoast.org/sensors/cco/penzance> ;
  ssn:observedProperty waves:Mean_Wave_Direction ;
  ssn:featureOfInterest coastD:PhysicalMetOcean ;
  ssn:observationResultTime _:time_1 ;
  ssn:observationResult _:res_1 .
_:time_1
  rdf:type w3time:Interval ;
  w3time:hasBeginning "2013-03-26T00:00:00"^^xsd:dateTime ;
  w3time:hasEnd "2013-03-26T00:30:00"^^xsd:dateTime .
_:res_1
  rdf:type ssn:SensorOutput ;
  ssn:hasValue _:val_1 .
_:val_1
  rdf:type ssn:ObservationValue ;
  ssnExt:hasQuantityUnitOfMeasure coastD:degree .
  ssnExt:hasQuantityValue "153.300"^^xsd:double .
```

Listing 5.1: A CCO observation represented in RDF Turtle notation

An example observation is shown in Listing 5.1. Each observation contains the URI of the sensor that made this measurement, the URI of the property it measures, the URI of the feature that this property is observed of, the time interval across which the observation was made, and finally, the measurement.

Table 5.1: Input datasets

	Date	No. of triples	No. of observations
Dataset1	2013-03-26T00:00/ 2013-03-26T00:30	2,366	169
Dataset2	2013-03-26T00:00/ 2013-03-26T03:00	14,098	1,007
Dataset3	2013-03-26T00:00/ 2013-03-27T00:00	110,572	7,898
Dataset4	2013-03-26T00:00/ 2013-03-31T00:00	462,560	33,040
Dataset5	2013-03-26T00:00/ 2013-04-06T00:00	1,121,974	80,141

For many of our experiments, we need to stress the system. The CCO update rate of nearly 2500 triples per half hour (seven observations from each of the 24 sensors) is not sufficient. Therefore, we have performed a number of SPARQL Construct queries over the CCO triple store and stored the results in different files. Table 5.1 shows the statistics of these datasets. One of them – dataset1 – simply contains all observations received over one half hour. Another one – dataset3 – contains observations of all sensors observed over a

whole day, comprising approximately 100k triples. The last one contains approximately 1 million triples and represents observations over a period of 11 days.

5.1.2 Functionality Tests

The main objective of these tests is to show that the system operators work accurately and produce the expected results for a range of typical use cases in weather sensor networks. We have defined a number of rules that range from simple pattern matching to more complex ones that require particular functionalities, such as aggregation, dealing with static and dynamic input, and ontology reasoning. For each of them, we describe the rule and motivation behind it, its RIF syntax, a simplified Rete diagram of it, and a sample of the produced output.

A source node at the bottom of the networks get real-time input streams via HTTP from CCO sensor URIs[16]. For simplicity, we only run the following use cases using input from five sensors, as we aim to assess the functionality of the system here, not the performance. The source node reads these URIs every half an hour to get new observations, and sends it to the following nodes in the network.

Note that in the rules described in the following sections, the namespace prefixes are those presented in Listing 5.1.

5.1.2.1 Basic Pattern Matching

Alert when a wave height gets above a certain level.

Motivation: This is a basic but useful rule for monitoring applications. Detecting high waves can be very crucial for decision making by coastal flood managers. This rule tests the system's ability to match RDF triple patterns and join their bindings to produce the expected output. As the rule syntax below does not specify a local window size, the system uses the global window specified at the import section of the rule document. If also not specified, the system uses its default values.

[16] e.g., http://rdf.api.channelcoast.org/observations/cco/boscombe/Hs/latest

Rule:

```
Prefix( pred <http://www.w3.org/2007/rif-builtin-predicate#> )
Forall ?ob ?v (
  If  And(?ob [rfd:type -> ssn:Observation]
       ?ob [ssn:observedProperty -> waves:Wind_Wave_Height]
       ?ob [ssn:observationResult -> ?result]
       ?result [ssn:hasValue -> ?value]
       ?value [ssnExt:hasQuantityValue -> ?v]
       External (pred:numeric-greater-than(?v 4.00)))
  Then ?ob [ssnExt2:alert -> ?v])
```

Rete network:

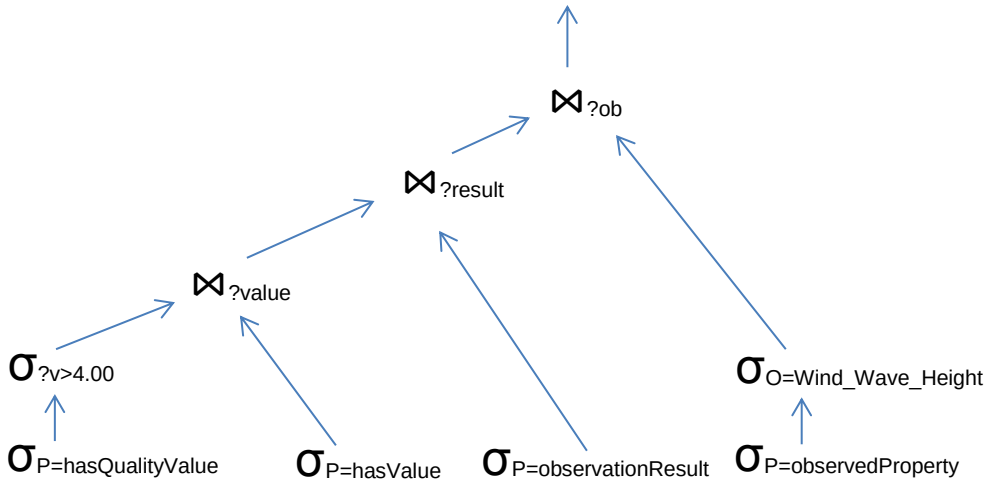


Figure 5.2: Rule 1 network

Example results:

```
http://id.api.channelcoast.org/observations/cco/perranporth/Hs/20130622#220000
ssnExt2:alert 4.05
http://id.api.channelcoast.org/observations/cco/perranporth/Hs/20130622#230000
ssnExt2:alert 4.29
http://id.api.channelcoast.org/observations/cco/perranporth/Hs/20130622#233000
ssnExt2:alert 4.38
```

5.1.2.2 Aggregation

Find the half daily average of wave heights for a specific sensor.

Motivation: Aggregates are very important figures, especially in streaming applications. In this use case, we test the system's ability to calculate averages over a specified time window for a specific sensor.

Rule:

```
Prefix( func <http://www.w3.org/2007/rif-builtin-function#> )
Forall ?ob ?v ?avg(
  If  (And( ?ob [ssn:observedProperty -> waves:Wind_Wave_Height]
       ?ob [ssn:observedBy ->
http://id.api.channelcoast.org/sensors/cco/minehead]
       ?ob [ssn:observationResult -> ?result]
       ?result [ssn:hasValue -> ?value]
       ?value [ssnExt:hasQuantityValue -> ?v]
       ?avg = External (func:numeric-avg(?v)) ) 12 h
  Then ssnExt2:Average [ssnExt:hasQuantityValue -> ?avg])
```

Rete network:

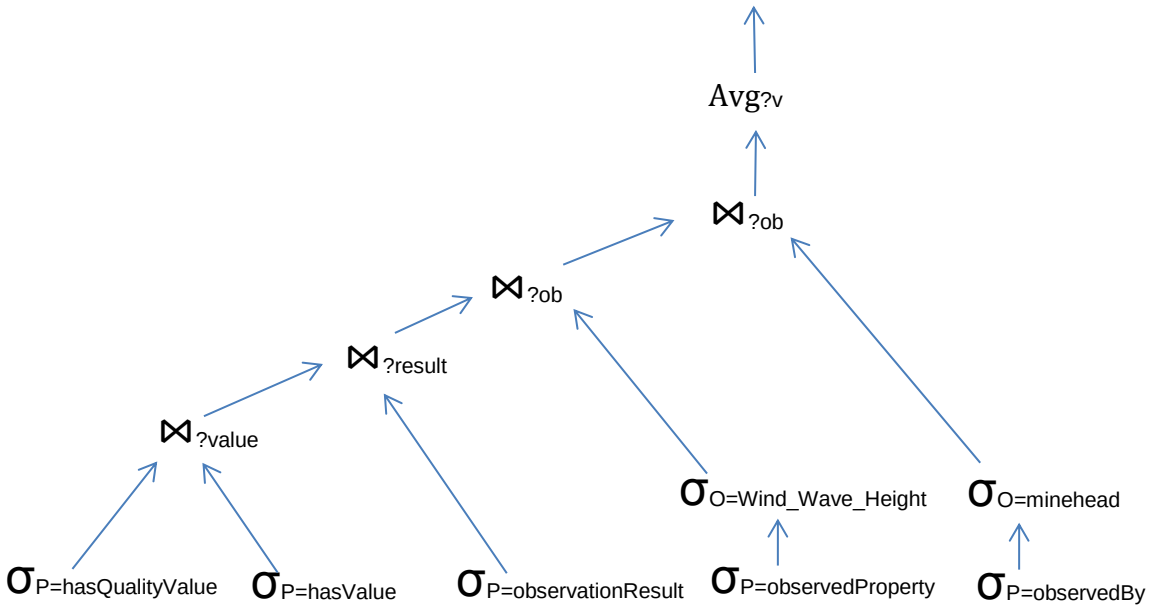


Figure 5.3: Rule 2 network

Example results:

```
ssnExt2:Average ssnExt:hasQuantityValue 0.8200000000000002
```

5.1.2.3 Static and Dynamic

Find the location of sensors that observe high waves.

Motivation: This use case is very similar to the first one. However, it also shows the coordinates of the sensors that observed the output. These coordinates can then be further used by other rules to locate nearby places of interest. As the locations of sensors are

typically static, this rule gets input from a static file containing the locations of all sensors and receives dynamic observation updates like the previous rules.

Rule:

```
Prefix (sw <http://sweet.jpl.nasa.gov/2.1/sweetAll.owl#>)
Forall ?ob ?v ?co1 ?co2(
  If  And( ?ob [ssn:observedProperty -> waves:Wind_Wave_Height]
       ?ob [ssn:observedBy -> ?s]
       ?ob [ssn:observationResult -> ?result]
       ?result [ssn:hasValue -> ?value]
       ?value [ssnExt:hasQuantityValue -> ?v]
       External (pred:numeric-greater-than(?v 4.00))
       ?s [ssn:hasDeployment -> ?dep]
       ?dep [ssn:deployedOnPlatform -> ?plat]
       ?plat [sw:hasLocation -> ?loc]
       ?loc [sw:coordinate1 -> ?co1]
       ?loc [sw:coordinate2 -> ?co2])
  Then  And( ?ob [ssnExt2:alert -> ?v]
    ?ob [ssnExt2:atCoordinate1 -> ?co1]
    ?ob [ssnExt2:atCoordinate2 -> ?co2]))
```

Rete network:

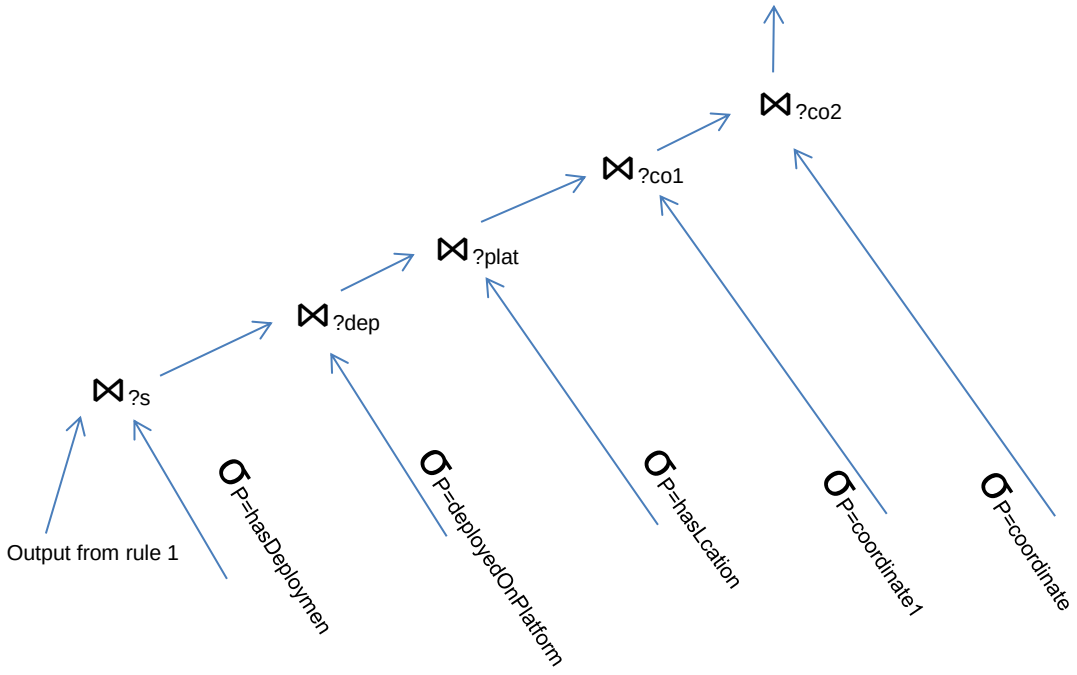


Figure 5.4: Rule 3 network

Example results:

```
http://id.api.channelcoast.org/observations/cco/perranporth/Hs/20130622#220000
ssnExt2:alert 4.05
http://id.api.channelcoast.org/observations/cco/perranporth/Hs/20130622#220000
ssnExt2:atCoordinate1 50.35
http://id.api.channelcoast.org/observations/cco/perranporth/Hs/20130622#220000
ssnExt2:atCoordinate2 -5.18
http://id.api.channelcoast.org/observations/cco/perranporth/Hs/20130622#230000
ssnExt2:alert 4.29
http://id.api.channelcoast.org/observations/cco/perranporth/Hs/20130622#230000
ssnExt2:atCoordinate1 50.35
http://id.api.channelcoast.org/observations/cco/perranporth/Hs/20130622#230000
ssnExt2:atCoordinate2 -5.18
```

5.1.2.4 Background Reasoning

Find the sensors that observe wind speeds higher than a known hurricane (inspired by a query in Zhang et al. (2012)).

Motivation: Extreme weather conditions can be detected by comparing observed values to historical data. This use case, like the previous one, involves dealing with static and dynamic data but also needs to perform background ontology reasoning. For the historical hurricane data, we prepared a small dataset taken from dbpedia, containing information about a number of hurricanes. In dbpedia, the class yago:Hurricane111467018 is used to describe a hurricane. However, this class has only two instances linked directly to it, while it has 88 subclasses that are used by other hurricane instances. For example, dbpedia:Hurricane_Fabian is of type yago:HurricanesInBermuDA, which is one of the yago:Hurricane111467018 subclasses. To be able to find information about all possible hurricane instances, we need to first perform background ontology reasoning. When the background reasoning mode is activated, rule 9 of the RDFS entailment rules will assert these hurricanes as instances of the hurricane superclass. These new assertions are inserted back as input so that the RIF rule can find them.

Rule:

```
Prefix (yago <http://dbpedia.org/class/yago/>)
Prefix (dbpprop <http://dbpedia.org/property/>)
Forall ?ob ?s ?hur (
  If  And( ?ob [ssn:observedProperty -> waves:wind_speed]
       ?ob [ssn:observedBy -> ?s]
       ?ob [ssn:observationResult -> ?result]
       ?result [ssn:hasValue -> ?value]
       ?value [ssnExt:hasQuantityValue -> ?v]
       ?hur [rdf:type -> yago:Hurricane111467018]
       ?hur [dbpprop:1MinWinds -> ?hurWind]
       External (pred:numeric-greater-than(?v ?hurWind)))
  Then ?ob [ssnExt2:alert -> ?s])
```

Rete network:

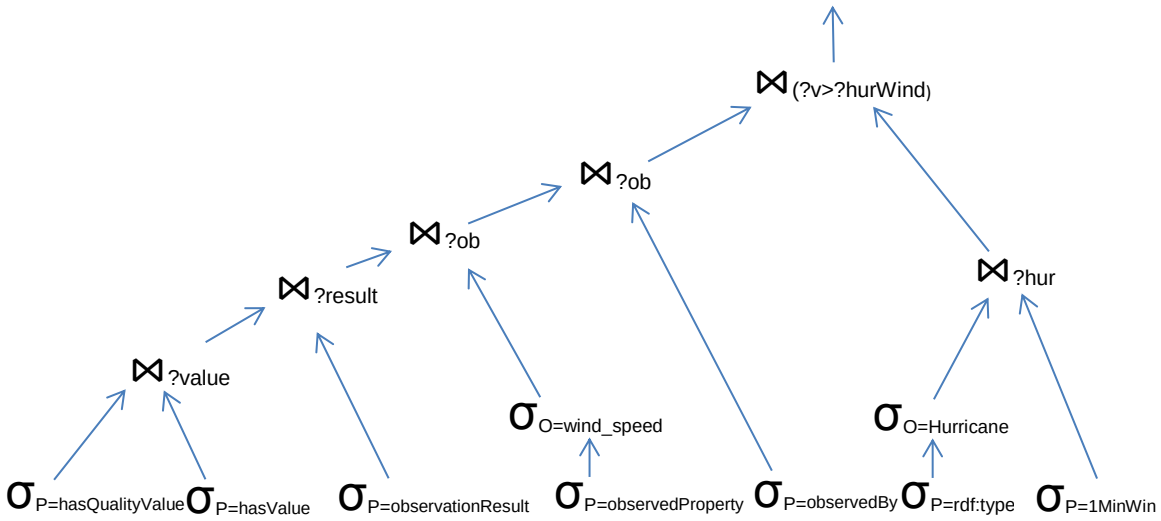


Figure 5.5: Rule 4 network

5.2 Comparative Evaluation

To evaluate the efficiency of our native semantic stream reasoning approach, we compare R4 to other semantic reasoners. First, we compare it with Jena, a static semantic reasoner, in terms of the throughput and processing time needed to process different datasets. Second, we compare R4 to Etalis and Sparkwave – stream processing engines that support background reasoning. All experiments were performed on an Intel Core i5 computer running at 3.2 GHz with 8 GB memory. Each experiment was performed five times, and the results presented in this chapter represent the average. The complete results of selected experiments can be found in Appendix B.

5.2.1 Comparing Stream Reasoning to Static Reasoning

In this experiment, we compare R4 to the Jena generic rule-based reasoner. Jena is an open source Semantic Web framework for Java. Its API enables reading, processing, and writing RDF graphs as 'Model' Java objects. It was chosen because of its rich Java library, which made it easy to check and compare results, control input rates, etc.

Jena provides a number of inference engines, including configurable RDFS, OWL reasoners, and a generic rule-based reasoner that supports user-defined rules. The generic reasoner's default configuration works in a 'hybrid' mode, in which it uses a forward

chaining Rete engine and a backward chaining engine in conjunction. It can also be configured to run the forward chaining engine only.

5.2.1.1 Methodology

As one of the most important requirements for any streaming application is to provide timely responses, we compare Jena's generic rule-based reasoner and R4 in terms of the processing time needed to reason over the input data. Measuring processing time for Jena is straight forward, as it starts and ends processing at specific points of time, where all results are provided at the end when the engine has finished processing all the input. However, R4 performs continuous reasoning, where results are produced incrementally as soon as they are obtained, and the rule has no explicit end time. Therefore, we take the latter of two measures: the time when the last output result is produced and the time when the last input triple is processed.

We used a similar rule to the one in 5.1.2.1 but with background RDFS reasoning applied. We prepared a simple synthetic schema of five observation classes (Observation1 to Observation5), where each class is a subclass of the next, and assigned the class Observation as a subclass of the first one. The rule then asked for observations of type Observation1 with high waves.

5.2.1.2 Experiment 1

The naïve way to compare R4 and Jena is to push the whole input dataset to each engine, apply the same rule, and compare the processing times. This way, Jena receives the whole dataset at once and performs static reasoning one time. On the other hand, R4 receives the input data as a finite stream. It observes it through a time window and incrementally performs reasoning. We conducted this experiment using the five datasets described in Table 5.1 (on page 89) and chose 100 milliseconds as R4's default window size.

Nevertheless, this experiment is unfair for both Jena and R4. It is unfair for Jena, as R4 does not work on the whole dataset at any time instance, while Jena has to consider every triple. On the other hand, it is unfair for R4, as Jena does not perform any retractions, while R4 continuously checks and removes outdated elements. Therefore, we conducted another experiment in which we tried to mimic a streaming situation.

5.2.1.3 Experiment 2

In this second experiment, the input file was partitioned to chunks of specific size, forming what we call 'updates'. We periodically sent one update to Jena and R4 until the last chunk. To help Jena remove expired elements, we saved each update in a separate model and forced the reasoner to remove the elements of this model when we considered them expired (as per the time window specified for R4). For example, if we send an update every second, and the time window is five seconds, at the sixth iteration, we insert the sixth update and remove the first update from the inference model, and so on. After each update, we measured delay (or response time), which is the time taken by each engine to process the whole update (including insertion and removal for Jena).

We ran this experiment using different update and window sizes. We used dataset 3 from Table 5.1, which contains observations from all sensors over a whole day as the input dataset. We partitioned it into 48 sets; each contains observations of half an hour (2,300 triples). We also partitioned it into 24 sets with observations of a whole hour for each (4,600 triples). Different window sizes are used to reflect different ratios of change. We also used dataset 4 from Table 5.1, partitioned into ten sets. Table 5.2 describes seven settings for the second experiment.

Table 5.2: All different settings in experiment 2

No.	Dataset size	Update size	Window size (hours)	Change percentage
1	110,572 triples 24 hours	2,300 triples Half-hour	10	5%
2			5	10%
3		4,600 triples 1 hour	10	10%
4			5	20%
5			2	50%
6			1	100%
7	462,560 triples 5 days	50,000 triples ~11 hours	44	25%

5.2.1.4 Comparative results

We performed the first experiment for Jena with the default configuration (default Jena), Jena with the forward Rete setting (Rete Jena), and R4. The average processing times of reasoning over different datasets are presented in Table 5.3 and illustrated in Figure 5.6. We also derived the input throughput of each system by dividing the processing time by the dataset size, depicted in Figure 5.7. R4 outperforms the default Jena in all cases and

outperforms Rete Jena in most of them. The figure shows that, while they perform comparably with small datasets, the difference becomes clear with big datasets. Jena spends considerable time building in-memory graphs and performing full reasoning over them, while R4 continuously matches patterns on the fly.

Table 5.3: Processing time results for the experiment 1 (pushing the whole dataset)

	Processing time (in seconds)		
Input dataset	**Default Jena**	**Rete Jena**	**R4**
Dataset1	0.29	0.24	0.13
Dataset2	0.49	0.42	0.37
Dataset3	1.73	1.44	1.68
Dataset4	5.66	4.79	3.98
Dataset5	14.43	12.85	8.03

We note that R4 is expected to produce less or incomplete results compared to Jena, as it does not consider the full dataset at any time instance. However, this was not the case due to the nature of the applied rule. The rule does not join triples from different graphs (observations) that may potentially arrive in a time interval that is bigger than the time window. All joins in the rule are intra-graph, and as the graph size is small (14 triples) and the input throughput is high, the graph always fits in the same window.

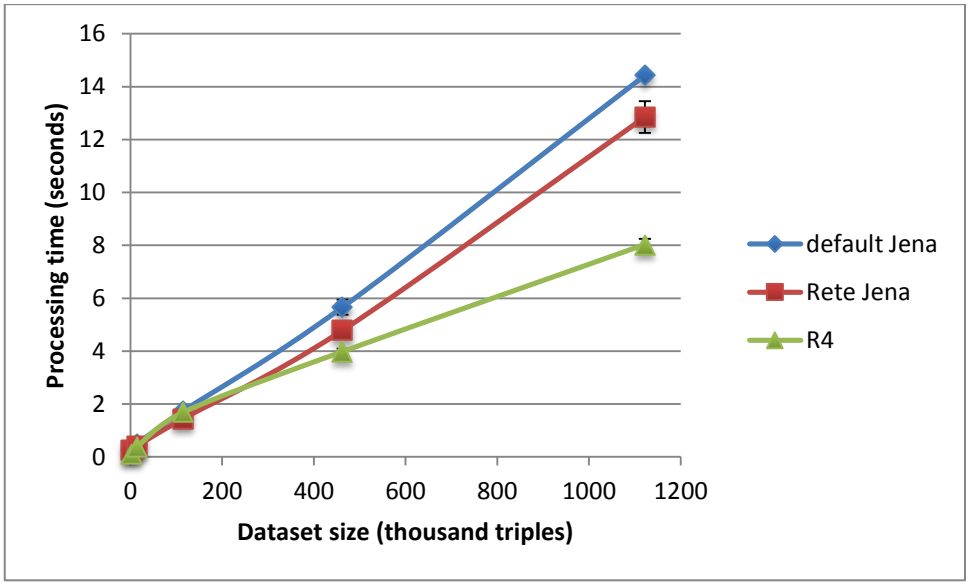


Figure 5.6: Processing time results for the experiment 1 (see table 5.3)

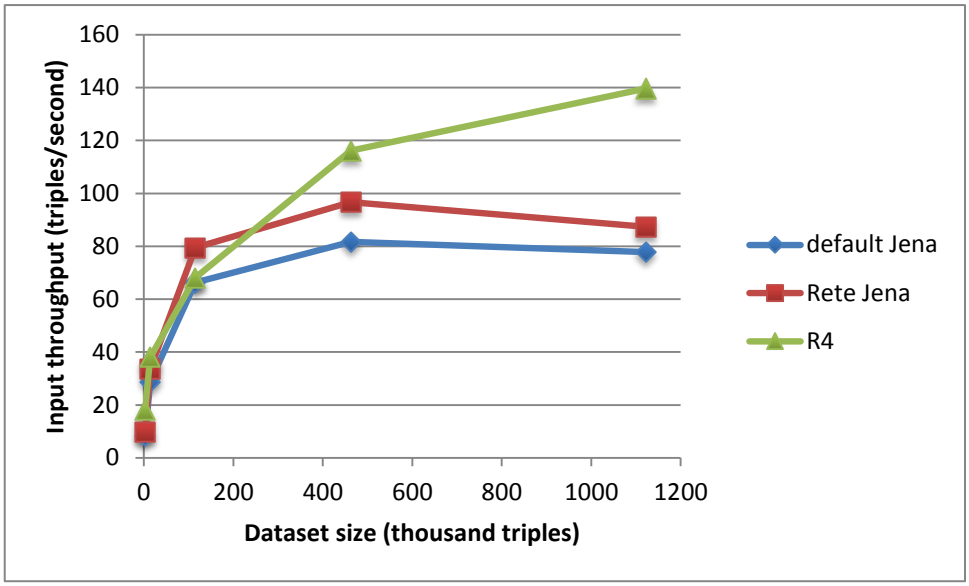


Figure 5.7: Throughput results for the experiment 1

We also performed the second experiment for the default Jena, the forward chaining Rete Jena, and R4 using the different update and window sizes described in Table 5.2.

For the first setting, 2300 triples are fed every half hour; Figure 5.8 shows the response time of each system when the window size is set to ten hours[17]. We notice that both Jena settings perform similarly during the first ten hours or 20 updates because there is only insertion at this stage. At the 21st update, the windows become full, and the systems have to start removing expired elements. Here, the cost of the default Jena suddenly increases, while Rete Jena and R4 keep on steady response rates. This is expected, as the default Jena re-performs the reasoning process for the whole data in the window for every removal. The Rete-based Jena and R4, on the other hand, update their entailments incrementally. R4 still outperforms Rete Jena, as its internal data structures are optimised for the time-based removal of expired data (unlike Jena, which has to search for these data in order to remove them).

In the above experiment, each update only changes 5% of the valid data, which justifies the huge difference between the incremental systems (R4 and Rete Jena) and the default Jena that re-computes entailments of the valid data from scratch. We changed the window size to five hours, so each update now changes 10% of the valid data. The results of this setting are depicted in Figure 5.9. It shows a similar pattern, but the default Jena has closer results to the other engines.

[17] To avoid the long running time, we actually insert every thirty seconds; thus, the window size is set to 10 minutes.

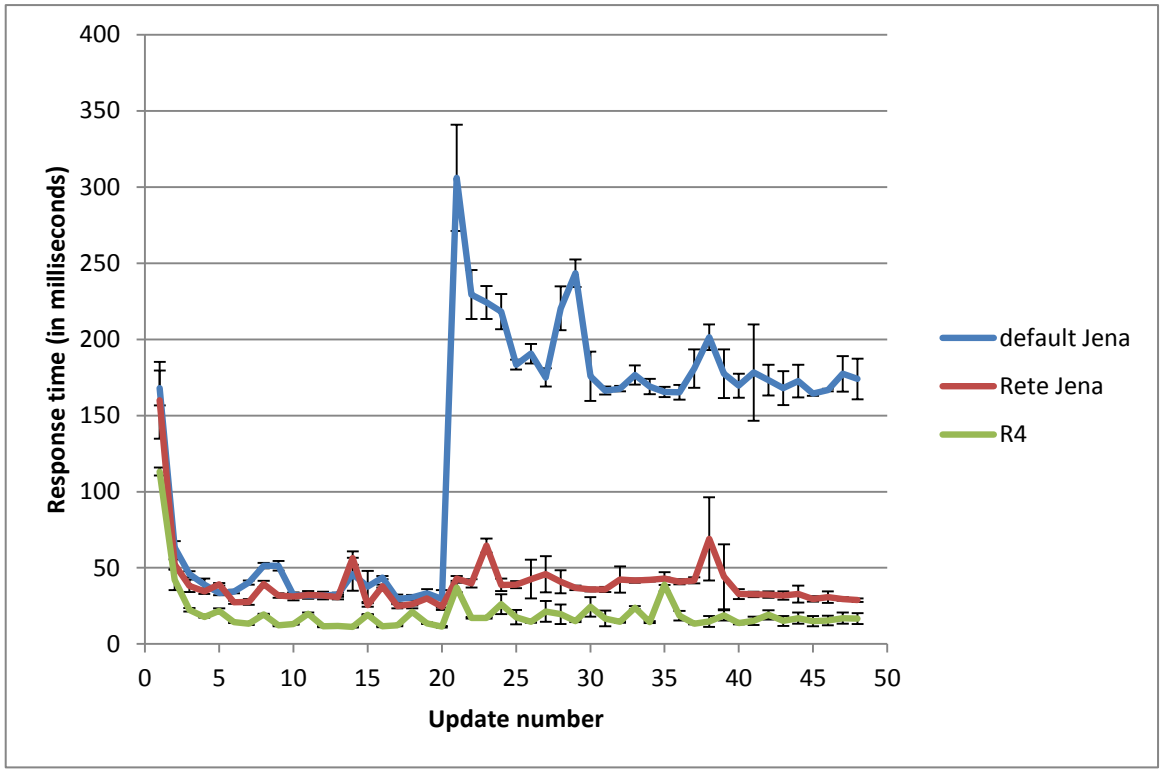


Figure 5.8: Response time for experiment 2, setting 1 (24 hours worth of data, updated every half an hour, window size 10 hours)

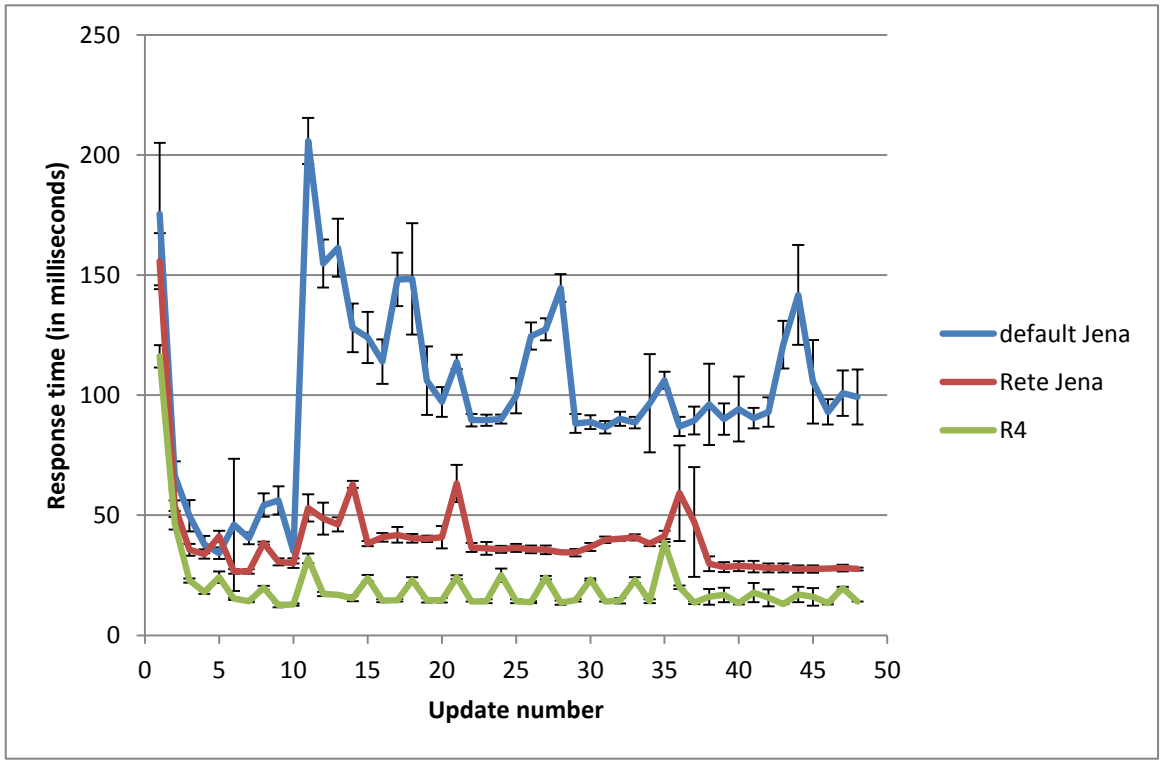


Figure 5.9: Response time for experiment 2, setting 2 (24 hours worth of data, updated every half an hour, window size 5 hours)

We ran the same experiment again with the second partitioning, in which each update contains 4600 triples, representing a whole hour worth of data. We varied the window size between ten hours (10% change), five hours (20% change), two hours (50% change), and one hour (100% change). Figures 5.10–5.13 show the response times of the three engines. The same trend is noticed, and for the last setting, both Jena engines become very close to each other, as the whole dataset is changed at each update.

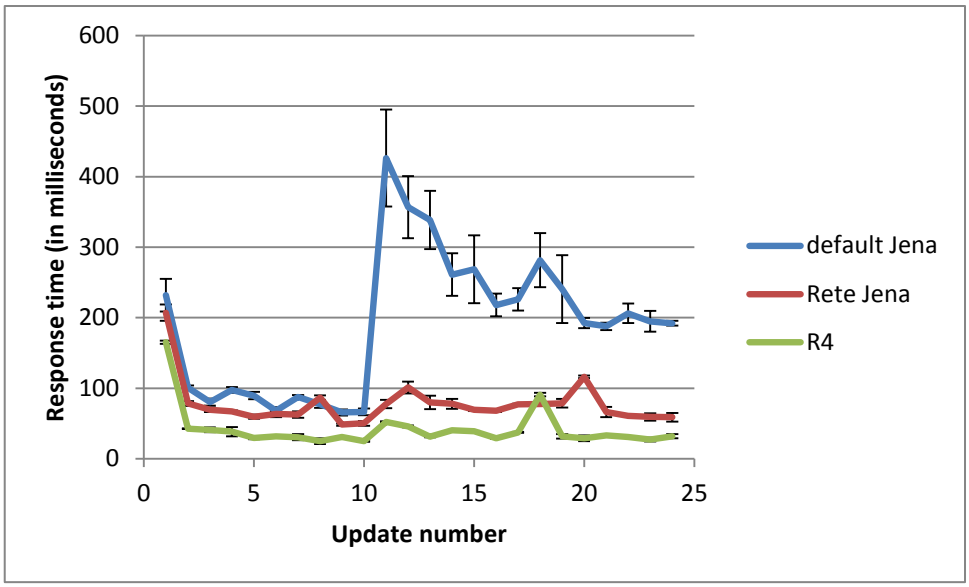


Figure 5.10: Response time for experiment 2, setting 3 (24 hours worth of data, updated every hour, window size 10 hours)

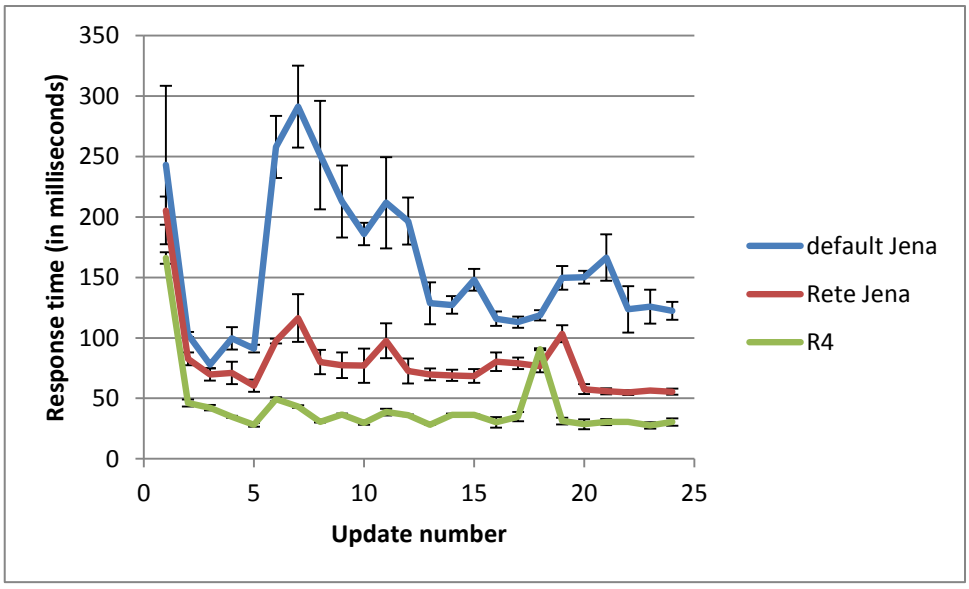


Figure 5.11: Response time for experiment 2, setting 4 (24 hours worth of data, updated every hour, window size 5 hours)

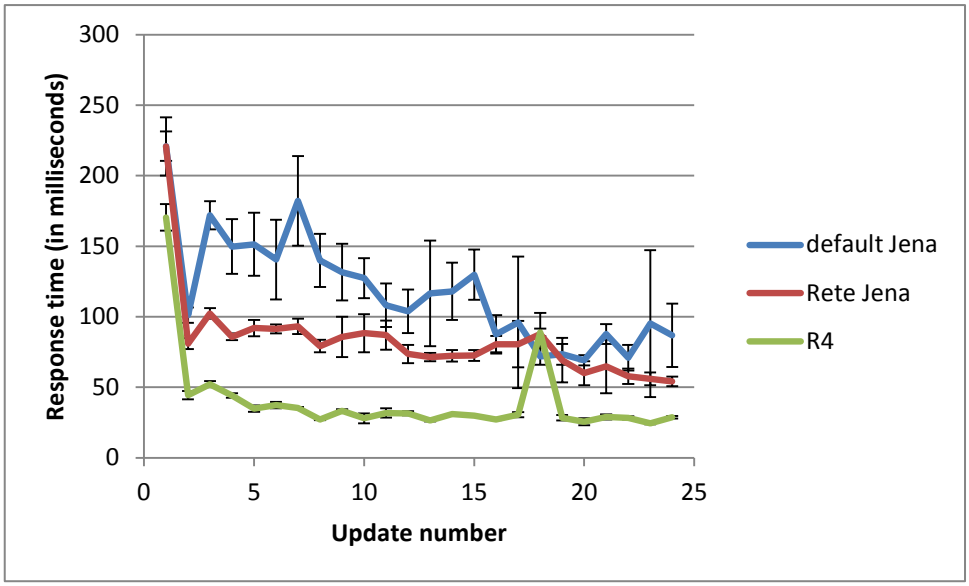


Figure 5.12: Response time for experiment 2, setting 5 (24 hours worth of data, updated every hour, window size 2 hours)

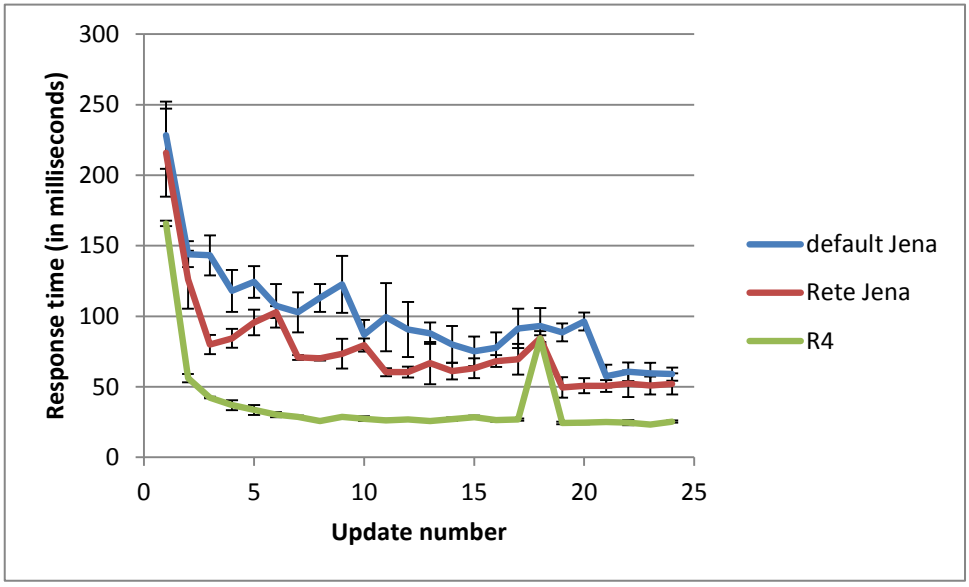


Figure 5.13: Response time for experiment 2, setting 6 (24 hours worth of data, updated every hour, window size 1 hour)

In the last setting, we changed the input file to the fourth dataset from Table 5.1 – containing almost half a million triples – to see if these results hold with bigger updates and windows. In this setting, each update inserts 50,000 triples, and the window size is set to include 200,000 triples. Figure 5.14 illustrates the results of this experiment, which shows that R4 still has the lowest response time, followed by the Rete Jena, while the default Jena comes last as in previous settings.

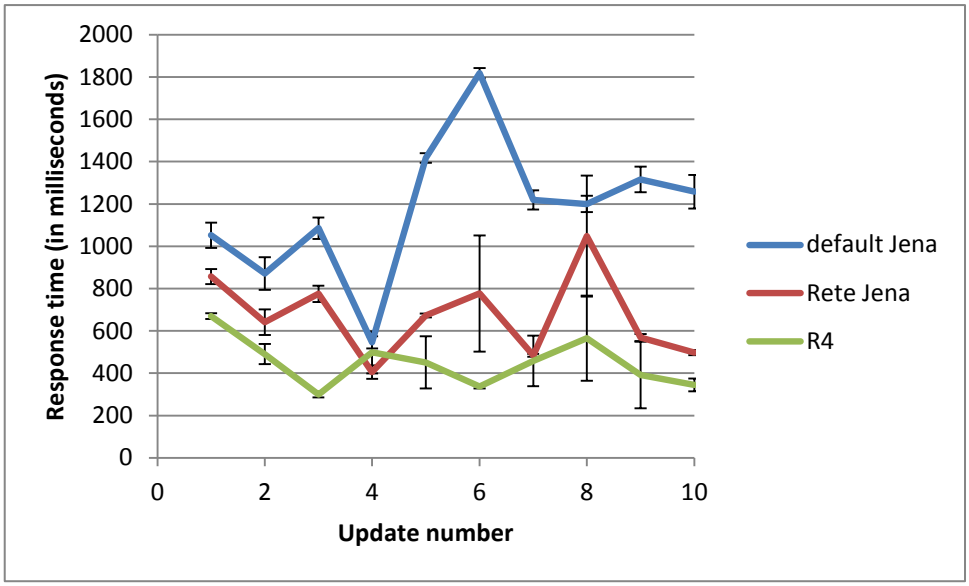


Figure 5.14: Response time for experiment 2, setting 7 (5 days worth of data, updated every ~11 hours, window size ~44 hours)

For each setting, we calculated the average response time of each engine, taking into account the iterations after the window is full. The results are presented in Table 5.4. For all of the above settings, R4's average response time is almost half of the response time of the Rete-based Jena and varies between 10–33% of the average response time of the default Jena depending on the percentage of change.

Table 5.4: Average response time for all settings in experiment 2

		Setting		Default Jena		Rete Jena		R4	
No.	Update size	Window size (hours)	Change %	Average response time (ms)	Standard deviation	Average response time (ms)	Standard deviation	Average response time (ms)	Standard deviation
1	2,300 triples, half hour	10	5%	188.59	32.01	39.61	9.22	18.89	6.30
2		5	10%	110.89	27.08	38.29	9.53	17.83	5.63
3	4,600 triples, 1 hour	10	10%	256.50	72.53	76.36	15.88	39.27	16.47
4		5	20%	168.25	54.25	76.04	17.50	36.77	14.17
5		2	50%	114.11	33.13	77.55	13.41	34.29	13.78
6		1	100%	94.77	24.93	70.59	19.05	31.70	13.59
7	50,000 triples, 11 hours	44	25%	1371.23	232.46	674.60	213.88	424.57	85.62

5.2.2 Comparing to State-of-the-art Stream Reasoning Systems

To evaluate the performance of R4, we compared it with state-of-the-art stream reasoning systems that provide the capability of performing lightweight background reasoning on streamed semantic data (reviewed in Chapter 3, Section 3.2.1). These included Etalis (Anicic et al., 2012), Sparkwave (Komazec et al., 2012), Streaming knowledge bases (Walavalker et al., 2008), and the incremental reasoner presented in Barbieri et al. (2010b). To the best of our knowledge, the latter two implementations were never made public, so we have only compared R4 to Etalis and Sparkwave.

Similar to R4, Sparkwave uses Rete networks that work directly on RDF streams. However, Sparkwave's reasoning expressivity is limited to specific RDF Schema entailment rules (plus owl:inverseOf and owl:SymmetricProperty) while R4 is built to support general purpose rules that can be written in RIF Core. Rete networks in Sparkwave are used to process continuous SPARQL queries, while the schema entailments support is provided using an additional network called Ɛ–network that precedes the Rete network. Sparkwave pre-computes the schema closure and use it to build the Ɛ–network. This network encodes schema-driven property hierarchies with specified domain and range definitions connected to class hierarchies. However, this network is activated by single triples from the stream, as it treats its input in a stateless way. Therefore, as explained in (Dell'Aglio and Della Valle, 2014), Sparkwave cannot be extended to support RDFS+ as it cannot for example support the owl:transitiveProperty construct, because it needs to be activated by multiple triples from the stream. R4 on the other hand processes RDFS+ (encoded in RIF in Appendix A) in the same way it processes general purpose RIF rules, using Rete networks that can be activated by more than one streaming input. Furthermore, R4 enables re-entrancy, which enables a generated answer to re-enter the network to possibly participate in generating further answers.

In terms of stream models, Sparkwave uses point-in-time semantics for the time model, defining an RDF stream as an unbounded sequence of RDF triples associated with timestamps. In R4, an external RDF stream is an unbounded sequence of RDF graphs associated with timestamps. Internally, both R4 and Sparkwave use time intervals (only for beta memory elements in Sparkwave) but representing different semantics. In R4, an internal stream element is assigned an expiration timestamp based on the specified window, each generated partial or full result takes the latest start timestamp and the earliest expiration timestamp of it constituting elements to represent its validity, so that whenever

any constituting element expires, its generated result also expires. In Sparkwave, a generated partial or full result takes the earliest start timestamp and the latest start timestamp of its constituting element as a means to check if this interval falls in the specified window, while garbage collection is based on start timestamps only.

5.2.2.1 Methodological Considerations

Following the development of semantic stream processors, a number of benchmarks have emerged to test and compare their performance: SRBench (Zhang et al., 2012), CSRBench (Dell'Aglio et al., 2013), LSBench (Le-Phuoc et al., 2012c), and CityBench (Ali et al., 2015). While these benchmarks provide rich datasets, they mostly do not consider reasoning; with the exception of SRBench, none of the other benchmarks' queries require inference capabilities.

One of the most important measures of any stream processing system is the maximum input throughput, defined in Scharrenbach et al. (2013) as the number of data elements in the input stream consumed by the system per time unit; we use this as the key performance indicator in our comparative evaluation of the three systems. In addition, it is necessary to confirm that the data produced by a system is complete and correct according to its semantics.

5.2.2.2 Methodology

We set up Jtalis (the Java wrapper of Etalis) over SWI-Prolog v7.2.1 and installed Sparkwave v0.5.1.

We planned to run the same rule used in the comparison with Jena over the CCO dataset. However, Sparkwave was unable to read the dataset correctly as it does not parse blank nodes; this might be because it uses the hash-join algorithm to improve time efficiency, but builds the hashtables based on the URIs of the incoming data. Sparkwave also does not support data comparisons (needed for the greater-than condition in the rule). Therefore, we followed the experimentation strategy used by Sparkwave (Komazec et al., 2012) as it also requires reasoning over background knowledge to correctly answer the query. In this experiment, we use a synthetic dataset generated with the Berlin SPARQL Benchmark (BSBM) (Bizer and Schultz, 2009) containing 1.1 million triples (representing 100,000 limited time offers made available by an online market place). For background knowledge, we generated a small schema that described 329 product types arranged in a four-level hierarchy.

```
Prefix(dc <http://purl.org/dc/elements/1.1/>)
Prefix(bsbm_voc <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/>)
Prefix(bsbm_inst <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/instances/>)
Forall ?offer ?product ?vendor ?price ?from ?to ?delivery ?webpage(
  If And( ?offer [rdf:type -> bsbm_voc:Offer]
     ?offer [bsbm_voc:product -> ?product]
     ?offer [bsbm_voc:vendor -> ?vendor]
     ?offer [bsbm_voc:price -> ?price]
     ?offer [bsbm_voc:validFrom -> ?from]
     ?offer [bsbm_voc:validTo -> ?to]
     ?offer [bsbm_voc:deliveryDays -> ?delivery]
     ?offer [bsbm_voc:offerWebpage -> ?webpage]
     ?offer [dc:publisher -> ?publisher]
     ?offer [dc:date -> ?date]
     ?product [rdf:type -> bsbm_inst:ProductType73] )
  Then ?product [bsbm_voc:offerPrice -> ?price] )
```

Listing 5.2: A RIF Core rule that entails the offer price for all products of a specific type that are on offer

Each system was configured with the generated schema, a rule inspired by a query from the BSBM, and a window size (0.1, 1, 2, 5, and 10 second sliding windows are tested). The rule we used in this experiment, shown in Listing 5.2, is inspired by the Berlin SPARQL benchmark and entails the offer price for all products of a specific type that are on offer. As offers may be associated with product super-types, some background reasoning is needed to determine the offer price for specific product sub-types.

We then measured the time it took for the system to consume all 1.1 million triples for each configuration of each system, from which we can calculate the maximum throughput and average latency of that system for each window size. We also checked the correctness of results and found that all three systems provide the same correct results. This is because the operational semantic of the systems are similar. In terms of the SECRET model – described in (Dell'Aglio et al., 2013) – the report strategy of all systems are content-change, the tick is tuple-driven, and the output operator is Istream. Above that, the query used in our experiments does not match triples from different graphs, which means windowing mechanisms does not affect the results.

5.2.2.3 Comparative Results

Firstly, we note that the rule presented in Listing 5.2 takes significantly longer to process in Etalis (over three hours for the 1.1 million dataset with a one-second time window compared to less than two minutes for the other two systems). In order to further investigate this difference, we tried changing the And into a series of seqs (Anicic et al., 2011), which resulted in the system running 20x times faster (7.5 minutes for the same

test). Despite being semantically different[18], we would not expect such a drastic difference in running time unless Etalis were heavily optimised for the seq operator. This is likely, as Etalis is intended for event processing rather than continuous querying/reasoning, making the order of arrival of triples more relevant than their simple coincidence in the system.

In order to present graphics that provide a meaningful comparison between the time efficiency of each system with regards to changing window ranges, as in Figure 5.15, we chose to eliminate the results for Etalis using And from our dataset but include those using the rule modified to use seq when evaluating Etalis. It should be noted that we recognise that this modified rule is not semantically equivalent to that by which we evaluate R4 and Sparkwave, which cannot express the seq operator, but is suffcient to contrast the effect of the window range on the time effeciency of the three systems. Table 5.5 presents the processing times of both settings in Etalis ('And' and 'seq') in addition to Sparkwave and R4, for the rule presented in Listing 5.2, with a third-level product type (according to the four-level hierarchy schema). The schema is loaded first. The data is then provided to each system by specifying the file name from which it is supposed to be read, so that each system starts reading and processing the data from that file at the fastest rate at which it is able. We measure the time needed to process the whole dataset, starting from receipt of the first tuple (thus, the time required to process the schema is not included). For each system, we ran the same rule with the same dataset five times. The average processing time is presented. No warm up period was used to take account of disk caching.

Table 5.5: Comparing processing times (in seconds) of R4, Sparkwave, and Etalis

System	Window size (seconds)				
	0.1	1	2	5	10
R4	8.82	12.25	14.33	15.25	17.30
Sparkwave	27.05	74.48	104.37	163.82	228.81
Etalis (seq)	461.91	446.56	447.98	447.48	449.74
Etalis (And)	11587.32	11310.25	11567.61	11594.24	11496.49

As shown in figures 5.15 and 5.16, R4 is significantly faster than both Etalis and Sparkwave for the tested window sizes. Interestingly, while the maximum throughput

[18] For example, a query that asks whether two persons arrived at the same location within a five-minute interval using 'and' will produce results regardless of which person arrived earlier. Using 'seq' on the other hand, will only produce results if person1 in the query arrived before person2 in the specified time window.

decreases and processing time increases (apparently asymptotically) with window range similarly in both Sparkwave and R4, those of Etalis using seq appear to remain approximately constant for all window ranges tested. It should be noted that this behaviour of Etalis remains in the case where the query uses And. However, despite its constancy, the throughput and processing latency of Etalis are so dramatically lower/higher than the other systems, respectively, that we project that R4 will remain faster than both Sparkwave and Etalis in cases where the window sizes are orders of magnitude larger than those tested.

We note that the aim of this experiment was not to reproduce Sparkwave's experiment described in (Komazec et al., 2012), but was rather guided by Sparkwave's experiment; we aimed to carry out a like-for-like comparison between the three systems. However, we acknowledge that there is a discrepancy between the performance of Sparkwave in our experiment and its performance in published experiment (Komazec et al., 2012). While we have not used exactly the same dataset used in their experiment (as it was not provided by Komazec et al), we generated a dataset in the same way as they described in the Sparkwave experiment. Our generated dataset is smaller than in Sparkwave experiment, but of a roughly comparable size (1.1 million triples in our experiment vs. 2.2 million triples in the Sparkwave experiment). We used the same rule as in the Sparkwave experiment, but as they did not specify the target product type they used in the rule, we picked an arbitrary product type. We used the same schema, but with fewer subclasses of the chosen product type (Komazec at el note that the number of subclasses of the target product type had a negligible effect on throughput).

On a smaller problem, with a faster computer (3.2 GHz vs. 2.66 GHz) with more memory (8 GB vs. 4 GB RAM), Sparkwave performed worse in our experiment than it did in the Sparkwave experiment. For example, using a 5 second window, Sparkwave's throughput was 7,000 triples/second in our experiment, compared to 60,000 triples/second in their experiment. However, the description of the Sparkwave experiment in (Komazec et al., 2012) is incomplete, as it does not describe the number of results generated by the rule, which might have an effect on throughput. Therefore, it may not be reasonable to strictly compare the results of this experiment with that of Sparkwave experiment.

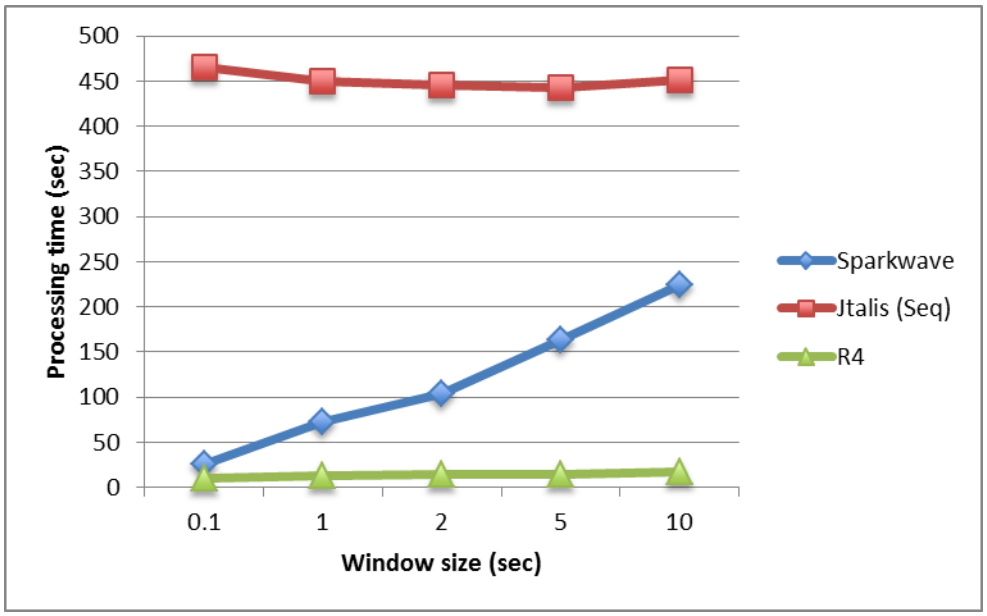


Figure 5.15: Processing time of R4, Sparkwave, and Etalis

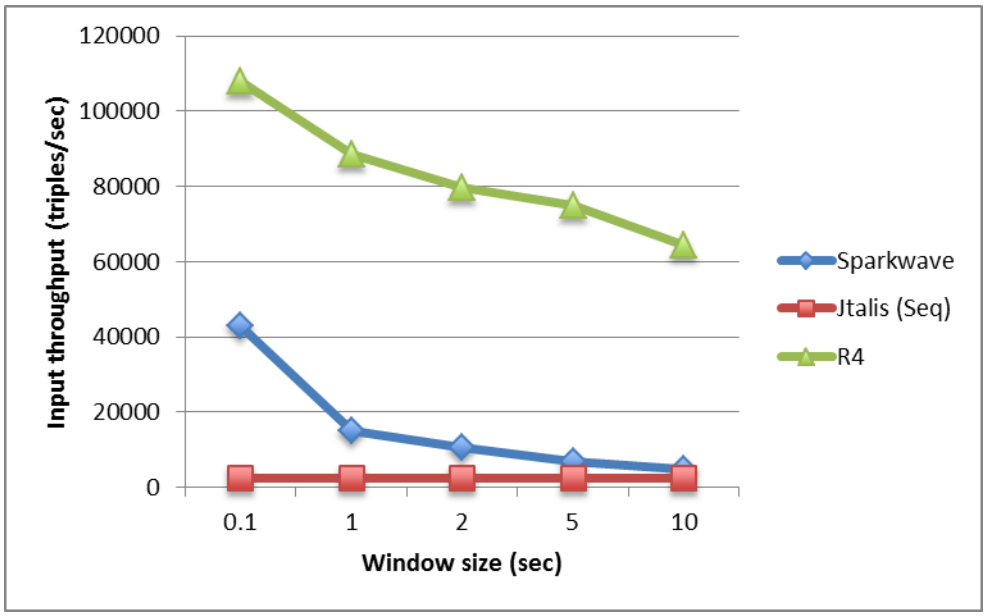


Figure 5.16: Input throughput of R4, Sparkwave, and Etalis

Finally, we also repeated the same experiment using different schemas with different number of subclasses, ranging between 40 and 1100 subclasses. All the three systems were not affected by the bigger schema size, showing similar performance to the original setting. Results of this experiment can be found in Appendix B.

5.3 Conclusion

This chapter has presented several evaluations of R4. A number of functionality tests have proved some of its capabilities, including basic pattern matching, aggregation, combining

static and dynamic data, and performing background reasoning. Comparing R4's performance with Jena's static reasoner positively supports our first hypothesis:

Hypothesis 1: It was anticipated that our continuous reasoning approach would improve throughput and responsiveness, when compared to a traditional static reasoner.

Furthermore, in terms of input throughput, R4 outperformed two of the state-of-the-art stream reasoning engines, namely Sparkwave and Etalis. However, our experiment is limited to only one rule using one dataset; the performance for other datasets with different characteristics and other rules might be different. However, if we were to use more than one rule, we might not necessarily be able to compare the results from one rule to the results from another rule. Therefore, a benchmark based on a selection of rules is needed, but it will also be limited to one dataset. While Sparkwave evaluated their system using three different rules, they reported consistent performance between all three rules (Komazec et al., 2012).

Experiments described in this chapter were run in R4 using statically optimised plans. With the addition of the adaptive optimiser – introduced next chapter – we expect R4 to highly outperform the other systems, especially in dynamic environments.

Chapter 6: Optimisation

As a single rule can usually be evaluated using different equivalent plans, an optimiser is needed to identify and generate the most efficient plan. In this chapter, we address the optimisation problem in the context of semantic stream processing. We firstly describe how R4 generates the initial plan based on simple heuristics in Section 6.1. We then discuss how the adaptive optimisation paradigm, introduced in the data stream management community (reviewed in Chapter 2, Section 2.1.3), is applied in R4 (Section 6.2). Section 6.3 then introduces the employed cost model, discussing several issues such as estimating selectivities and output rates. Monitoring, optimisation algorithms, and plan migration issues are discussed in sections 6.4, 6.5, 6.6, respectively.

6.1 Initial Rete Network Generation: Static Optimisation

In a database management system (DBMS), optimisers use statistics such as data cardinality and operator selectivities to build a cost model to choose the optimal plan at compile time (Garcia-Molina et al., 2000). However, these statistics are usually unavailable before runtime in a streaming context (Viglas and Naughton, 2002). Therefore, we use simple heuristics to generate a basic initial plan that should be refined at runtime after collecting sufficient statistics.

To generate the initial Rete network, the rule document is first parsed to identify individual rules and their condition elements (triple patterns). We use Squall's RIF Core parser[19] and extend it to handle stream and window specifications, using the EBNF given in Fig 4.7 and the code available from Squall.

The parsed ruleset is then passed to the optimiser, which starts by creating a source node for each input. Then, it handles the body – the 'If' part – of the ruleset by creating and connecting alpha and beta nodes. It forms the alpha network by creating an alpha node (with its own alpha memory) for each triple pattern. The beta network is then modelled by forming a left input adapter node to follow the first alpha node. For each other alpha node, a join node with a beta memory is created to connect it with the previous node in the beta network, which results in a left-deep beta network. At the end, the head – the 'Then' part – of the ruleset is handled by creating a terminal node with a template for every atom.

[19] https://github.com/sinjax/squall/tree/master/core/rif-core-parser

As the optimiser traverses the list of triple patterns, attempting to join them in the same order as they appear in the rule document, it enforces a known optimisation heuristic, which is to join alpha nodes with shared variables only (Garcia-Molina et al., 2000), leaving the triple patterns that have unique variables to the end. This optimisation technique aims to avoid Cartesian products, which are known to generate larger intermediate results (Mishra and Eich, 1992).

Before modelling any alpha or beta node, the set of already modelled nodes is checked. If an equivalent node – one that has the same condition, sources, and local window size if specified – has been already created, the old node (and its memory) is simply shared instead of creating a duplicate operator. This is one of the features of the original Rete algorithm (Forgy, 1982) that can save both memory and computational resources.

After dealing with all triple patterns in the ruleset, we handle the predicates. Predicates (or filters) are pushed as high as possible in the network as follows: If the predicate compares a variable to a static value, an alpha constraint is created at the first alpha node in which this variable appears. If the predicate compares two variables, a beta constraint is created at the first beta node where both variables appear in its tokens. Again, pushing filters as early as possible in the network is a known optimisation heuristic in DBMS (Garcia-Molina et al., 2000).

6.2 Adaptive Optimisation

As the initial plan was not constructed based on statistics of the input data, it is not expected to be optimal. Furthermore, the characteristics of data streams may continuously change, so the optimisation process needs to be adaptive. Apart from CQELS (Le-Phuoc et al., 2011), all other processors of semantic streams – reviewed in Chapter 3 – do not consider adaptive optimisation, as they basically depend on their underlying stream systems. CQELS, on the other hand, employs a white-box approach, so it implements its own optimiser. In the relational stream processing community, some systems supported adaptive optimisation using the plan-based approach, where the optimiser changes the running plan for all tuples (Babu and Widom, 2004; Cammert et al., 2008). Other systems used the routing-based approach, in which each tuple can follow a different route (Avnur and Hellerstein, 2000). CQELS supports routing-based adaptive optimisation using the Eddy operator (Avnur and Hellerstein, 2000). While eddies enable a fine-grained per-tuple optimisation, we argue that it would be very expensive for the RDF model. An RDF

document (or stream) consists of a large number of small triples, compared to the relational model, where a smaller number of records or tuples are composed, usually of more than three attributes. Holding statistics and choosing a route order for every small triple would cause a large computational overhead. We opt for more coarse adaptivity at an intra-query level as in (Babu and Widom, 2004; Cammert et al., 2008). In other words, R4's optimiser is plan-based, as opposed to the routing-based strategy used in CQELS.

The RDF model also causes more joins than the relational model, as a single triple does not hold much information. This makes it more crucial to have a good beta network topology. Therefore, due to the fact that we flattened the alpha network into single alpha nodes, we focus on optimising the join node ordering. The order of joins in the Rete network largely determines the performance of the plan, as join nodes are far more expensive than stateless filter nodes. While different equivalent plans should produce the same number of output results, different join orderings will cause a different size of intermediate results, affecting both processing costs and memory consumption, which ultimately mark throughput. A poorly-ordered plan might suffer even more, as stream characteristics change at runtime. The optimiser needs to ensure that the order of join nodes of the running plan is efficient for the current conditions.

Placing the more selective operators early in the network usually results in smaller intermediate results, which leads to faster processing and lower memory consumption. For example, assume we want to join three streams (A, B, and C), and each produces 10 triple/sec into a window of 10 seconds. Also, assume that there is a shared variable among all of them and that the join selectivity of $A \bowtie B = 0.1$, $A \bowtie C = 0.2$, and $B \bowtie C = 0.3$. As joins are commutative operators, there are three possible distinct ways to join the three streams: $(A \bowtie B) \bowtie C$, $(A \bowtie C) \bowtie B$, and $(B \bowtie C) \bowtie A$. The first join of the first plan will produce 100 triples every second, which means its beta memory size may grow up to 1000 triples. For the second and third plans, the first join's output rates are 200 triples/sec and 300 triples/sec, while its beta memory size are 2000 triples and 3000 triples, respectively. In this example, it is obvious that the first plan will outperform the other plans.

However, ordering joins based only on their selectivities does not always guarantee an optimal plan, as there are other factors that determine the performance of the network. In the example above, if stream A's input rate was 100 triples/sec instead of 10, output rates of the first join in the three plans would be 1000, 2000, and 300. This means that the first plan is no longer the optimal plan; instead, the third plan, which is ranked worst using the

lowest-selectivity-first heuristic, is now the best. Another important factor that affects the cost is the window size. Giving stream A a big window would have a similar effect to giving it a higher input rate. Therefore, a cost model that takes all these factors into account is needed. Using a cost model, the optimiser can estimate the costs of different possible orderings of joins and select the cheapest. The ultimate goal of the adaptive optimiser is to maximise throughput.

The adaptive optimisation process for continuous reasoning involves three parts of the system that work together to maintain good performance in changing streaming conditions. First, the monitor periodically tracks stream statistics, uses them to measure the cost of the currently running plan, and reports the cost to the optimiser. The optimiser compares the new cost to previously reported costs; if they significantly differ, it starts the re-optimisation process. Using a dynamic programming algorithm based on the cost model, it tries to find a cheaper plan. If a cheaper plan is found, it is communicated to the rule engine, and a plan migration is ordered. The rule engine then runs the chosen plan. This process is illustrated in Figure 6.1.

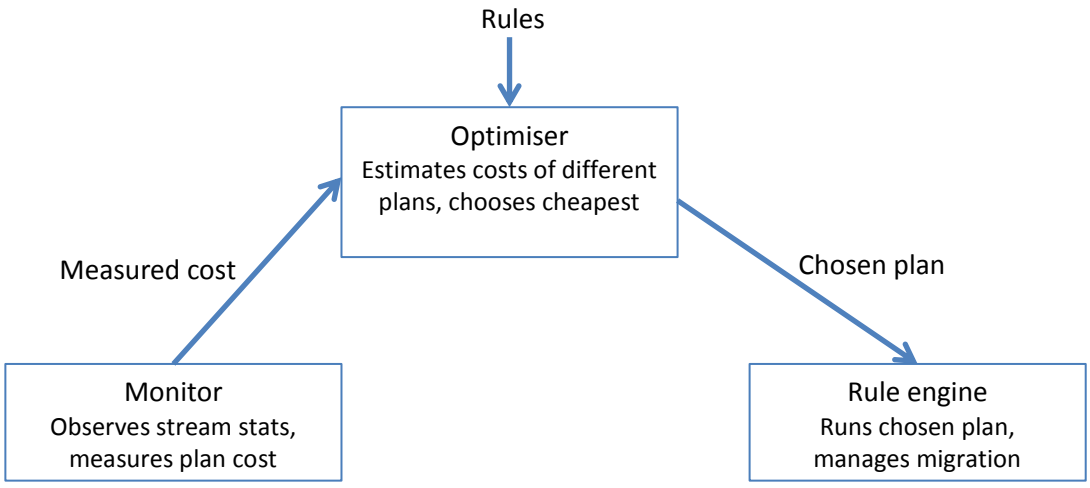


Figure 6.1: Adaptive optimisation process

In the next subsections, we first lay out the cost model used to estimate costs of join operators and plans. Then, the adaptation approach and its stages, including monitoring, re-optimisation, and plan migrations, are described in more detail.

6.3 Cost Model

Most optimisers in modern database systems are cost-based (Viglas and Naughton, 2002). Cost models are used to estimate costs of different equivalent plans in order to choose the most efficient. Some stream processing engines also use cost models instead of heuristics (e.g. (Viglas and Naughton, 2002; Cammert et al., 2008)). In streaming contexts, however, costs of operators are generally rate-based as opposed to relational, cardinality-based cost models (Schmidt, 2006).

As we are only interested in finding an efficient join ordering, and because joins are usually the most expensive operators in the network, we model the cost of the network as the summation of its joins' costs. We follow the research on estimating the cost of a windowed join – as in Kang et al., (2003), Ayad and Naughton (2004), and Cammert et al., (2008) – and use a per-unit-time cost model. In this model:

Cost of an operator = input rate * cost of handling one tuple

To model the cost of an individual join operator, we consider all the tasks that are performed by this operator. As described in Section 4.3.3, a join node that takes input from two windowed streams would perform the following tasks when a tuple is received from either side: the tuple is inserted into the corresponding window; the other window performs a garbage collection, which is then probed for matches; any results are then generated. Using L and R to refer to the left and right input streams to a join node, we represent the cost of a join operator as follows[20]:

$$C_{(L \bowtie R)} = C_{insert} + C_{invalidate} + C_{probe} + C_{result.}$$

In order to find the individual costs of these operations, we use the notions of input and output rates, window sizes, and selectivity. While these can be measured for running plans, we need a way to estimate them without actually creating and running the joins to avoid this unnecessary overhead. The following subsections explain how we calculate these different parameters in order to find each individual cost in the previous formula. Table 6.1 identifies the different parameters used throughout this section, where L and R denote the left and right input streams (as in the previous formula), and o refers to the output stream.

[20] Throughout this section, we use L and R to refer to the left and right input streams to a join node, and o to refer to the output stream.

Table 6.1: Cost model terms

λ_i	Arrival rate of tuples from a source input i
λ_L	Arrival rate of tuples for the left input of a join
λ_R	Arrival rate of tuples for the right input of a join
λ_o	Output rate of tuples from the current operator
W_L	Size of the left input window (number of tuples)
W_R	Size of the right input window (number of tuples)
W_o	Size of the output window (number of tuples)
B_L	Number of hash buckets in the left input window
B_R	Number of hash buckets in the right input window
f	Selectivity factor
C^I	Cost of inserting a tuple into a memory
C^H	Cost of evaluating the hash function of a single tuple
C^V	Cost of removing a tuple from a memory
C^E	Cost of re-organising the heap
C^P	Cost of a single evaluation of the join predicate
C^G	Cost of generating a new tuple
C^T	Cost of checking a tuple's timestamp
M	Size of a left tuple (number of triples comprising it)
$d_{S(a)}$	Number of distinct values of attribute a in stream S

6.3.1 Constant costs

First, we estimated the system-dependant costs (C^I, C^H, C^V, C^E, C^P, C^G and C^T), used in insertion, invalidation, probing, and result generation cost formulas, by measuring the actual time used by the CPU to perform each task. For example, we measured the time taken to hash 1000 tuples and divided the result by 1000 to determine the cost of a single evaluation of the hash function (i.e. C^P).

6.3.2 Estimating join's selectivity (f)

One of the most important variables to estimate in the cost model is the join's selectivity factor. In Babu et al. (2004), a profiler is used to estimate the selectivites of pipelined filters, where a periodic sample of the input is passed to different alternative paths to calculate the operators' selectivities. However, with the big number of joins expected using the RDF model, this can be quite expensive. We need a cheaper way that finds reasonably good estimates of joins' selectivities.

Cost models of windowed joins similar to ours – in Cammert et al. (2008) and Gomes and Choi (2008) – use the join selectivity factor without discussing how to estimate its value. In Ayad and Naughton (2004) and Getta and Vossough (2004), a join selectivity is described as a multiplication of the selectivities of all previous nodes in the network. This goes back to system R optimiser (Selinger, 1979), where the selectivity of (pred1) and (pred2), i.e. f (pred1⋈pred2), equals f (pred1) * f (pred2). However, because this assumes independence of the input values, using this with the RDF model results in severe underestimations of the number of output results of the join. RDF data usually demonstrate a high correlation between triples. For instance, an RDF triple <observation_x, observedBy, sensor_y> will always co-occur with a triple like <observation_x, observedProperty, property_z>. If we have 100 unique observations and are asked to join (?x, observedBy, ?y) with (?x, observedProperty, ?z), we expect the cardinality of the output results to be 100, as every observation contains information about the two predicates. However, using the above join selectivity estimation method underestimates the number of results to one: $f(_L\bowtie_R) = f(_L)*f(_R) = 1/100 * 1/100 = 1/10000$, so $W_o = W_L * W_R * f(_L\bowtie_R) = 100 * 100 * 1/10000 = 1$. On the other hand, it can be reasonable to assume the independence of observed temperature values from two different streams in a join like (?y, observedValue, ?x)⋈(?z, observedValue, ?x).

To address this problem, we apply a method based on an observation introduced in Neumann and Moerkotte (2011). They observe that many of the correlations between triple patterns stem from the fact that RDF uses multiple triples to describe the same entity, so searching for one of them is practically as selective as searching for all of them. In our graph-based RDF stream model, a stream is comprised of small graphs representing those entities, in the example above, weather observations. Therefore, we differentiate between the selectivities of joining triple patterns from a single graph – self-joins or intra-graph joins – and joining triple pattern from different graphs.

In the first case, the cardinality of joining multiple triple patterns from the same graph is equal to the number of graphs in the windowed stream, as they always co-occur. We consider triple patterns from one graph with bound objects as well as bound predicates. For example, joining (?x type Observation) with (?x observedValue >30) is an intra-graph join, but the selectivity of the first pattern is different than the second. While all 100 observations satisfy the first pattern, assume that only 10 observations satisfy the second. By making the join as selective as the most selective pattern, i.e. $f(_L\bowtie_R) = \min(f(_L),f(_R)) =$

$\min(1/100,1/10) = 1/100$, so $W_o = W_L * W_R * f(_L\bowtie_R) = 100 * 10 * 1/100 = 10$, which is the correct number of results.

In the second case, where joins happen between different graphs – either from the same stream or different streams – we assume independence. So, generally, for intra-graph joins, $f(_L\bowtie_R) = \min(f(_L),f(_R))$ and, for inter-graph joins, $f(_L\bowtie_R) = f(_L)*f(_R)$.

6.3.3 Estimating window sizes (W_o)

A join's right parent is always an alpha node. Hence, its window size can be measured at any time during the cost estimation phase, as they already exist and are not affected by the adaptive re-optimisation. In addition, it can be calculated using the following formula:

$$W_o(\sigma_a) = W_S * f(\sigma_a),$$

where W_S = the window length for tuple-based window, or $W_S = \lambda_i * T_S$, where T_S is the time-based window length.

On the other hand, the left parent of a join node is another join node, and its window size needs to be estimated since it might not be existent when the optimiser tries to investigate new ways to join the alpha nodes. If we can estimate the join selectivity factor, then the join's window size can simply be estimated as:

$$W_o = W_L * W_R * f(_L\bowtie_R).$$

While this formula appears exactly the same as the one used for finding the cardinality of a relational join – replacing window sizes with table cardinalities – it does actually take into account the temporal nature of sliding windows. This is explained by Ayad and Naughton (2004) as follows: A join's output window size is the number of valid tuples resulting from the join. A tuple stays valid as long as none of the tuples that comprise it have expired. If left and right input rates are steady and left and right window sizes at the beginning are W_L and W_R, the resulting window size for the already existing tuples in the windows will be $W_L * W_R * f(_L\bowtie_R)$. We consider tuples arriving from the left side; each arriving tuple produces $W_R * f(_L\bowtie_R)$ new tuples. However, we expect the same number of tuples in the resulting window to be invalidated as the earliest tuple in the left window becomes expired. The same logic applies to tuples arriving at the right side. Hence, the previous formula presents a good average estimate of the join results' window size.

6.3.4 Estimating output rates (λ_o)

In R4, as we use left-deep plans, right input rates are always the output rates of alpha nodes, while left input rates are the output rates of previous beta nodes. Output rates of alpha nodes are simply the output rates of raw streams multiplied by the selectivity of the filter predicate, as follows:

$$\lambda_o(\sigma_a) = \lambda_i * f(\sigma_a).$$

The stream output rate can either be known in advance or averaged at the source node. The selectivity factor of an alpha node can be defined as the percentage of tuples satisfying the filter predicate relative to the number of tuples input from the stream. As in database systems, it can be estimated as:

$$f(\sigma_a) = 1/d_{S(a),}$$

The number of distinct tuples can be measured online as the number of hash buckets of the corresponding alpha memory. Alpha memories are always available and are not affected by the re-optimisation process.

Finding the output rate of a join node is more complicated, as it has two inputs. We first find the output rate of a Cartesian product. In a Cartesian product node, every incoming tuple from the left parent is joined with every tuple in the right parent's window (and vice versa). Therefore:

$$\lambda_{o(L\times R)} = \lambda_L W_R + \lambda_R W_{L.}$$

Then, by defining the selectivity factor of a join node as the percentage of tuples satisfying the join predicate relative to a Cartesian product, we can estimate a join node's output rate as:

$$\lambda_{o(L\bowtie R)} = (\lambda_L W_R + \lambda_R W_L) f(_{L\bowtie R}).$$

Now, we discuss each individual cost in the previous formula – first in abstract terms, then in terms of our implementation.

6.3.5 Insertion cost (C_{insert})

Every time unit, the join node receives a number of tuples from both sides, equalling the output rate of its left and right parents. For each incoming tuple, there is a constant cost of adding the tuple to the corresponding memory. Therefore:

$$C_{insert} = (\lambda_L + \lambda_R) * C^I.$$

The constant cost is implementation dependant. As explained in Chapter 4, R4 uses a hash map and a priority queue – implemented as a binary tree – to model an alpha or beta memory. Therefore, C^I includes the cost of hashing the tuple, inserting it into a hash bucket, and adding it to the queue. While adding an element to a binary tree usually incurs a logarithmic cost in the length of the tree (Mehlhorn and Sanders, 2008), we found it to be constant in our experiments. Java's implementation of a priority queue adds the new element to the leaves and then uses the comparator to pop it up the tree until it finds its correct location. As the comparator orders tuples by increasing timestamps, and because incoming tuples usually arrive in order, they usually stay at the leaves – hence, the constant cost, as follows:

$$C_{insert} = (\lambda_L + \lambda_R) * (C^I + C^H).$$

6.3.6 Invalidation cost ($C_{invalidate}$)

Every time unit, the number of expired tuples in the left or right memory that need to be removed can be estimated to be equal to the output rate of the corresponding parent. This is especially accurate for tuple-based windows, in general, and time-based windows in steady state conditions where streams' output rate does not change often, as follows:

$$C_{invalidate} = (\lambda_L + \lambda_R) * C^V.$$

As opposed to the insertion cost in R4, the cost of deleting a tuple is actually logarithmic in the length of the tree, as it requires re-organisation of the queue. Whenever an element is deleted from the queue, which is usually the root of the tree (as it has the smallest timestamp), it is replaced by the last inserted tuple. The tree implementation then uses the comparator to push down this tuple until it finds its correct location, which should be in the leaves, as it has the biggest timestamp. This means it would be compared with a number of elements equal to the length of the tree. After deleting the stale tuple from the queue and re-organising it, the tuple should also be removed from the hash map, incurring a constant cost C^H. Therefore:

$$C_{invalidate} = (\lambda_L * \log_2(W_L+1) + \lambda_R * \log_2(W_R+1)) * C^E + (\lambda_L + \lambda_R) * C^H.$$

6.3.7 Probing cost (C_{probe})

When a new tuple arrives at one side, the join node searches the other side's window to find matches. The number of comparisons that needs to be done for each input depends on the join algorithm. For nested loop joins, every incoming tuple from the left side needs to be compared with each tuple in the right window (and vice versa), as follows:

$$C_{probe} = (\lambda_L W_R + \lambda_R W_L) * C^P.$$

For symmetrical hash joins used in R4, there is the constant cost of hashing the incoming tuple; the comparison cost will be a function of the number of tuples in the matching bucket. If we assume uniform distribution of tuples across buckets, each bucket will contain W_i/B_i tuples, as follows:

$$C_{probe} = (\lambda_L * W_R/B_R + \lambda_R * W_L/B_L) * C^P + (\lambda_L + \lambda_R) * C^H.$$

6.3.8 Result generation cost (C_{result})

If the searching process results in a match, there is a cost of generating the new result. While Kang's cost model (Kang et al., 2003) does not consider this cost, it is explicitly calculated in Cammert (2008) as a constant cost for each generated tuple, as follows:

$$C_{result} = \lambda_o * C^G.$$

However, in R4, the cost of creating a result is not simply a constant. There is the constant cost of creating the new tuple, and then, there is the cost of assigning its timestamp. This cost is found to be a function of the number of triples in the left tuple because we need to search them to find the latest start time and earliest end time stamps to assign them to the result. Thus, if M is the size of the left tuple:

$$C_{result} = \lambda_o * (C^G + M * C^T).$$

6.4 Monitoring

Every set amount of time, the monitor measures the cost of the currently running plan. It uses the same cost model presented in the previous sections, but as the plan is actually running, it uses measured window sizes and input rates instead of estimates. The monitor works in a bottom-up fashion, starting from the source nodes until the last join in the

network, as every join needs information about its parent's window sizes and output rates. At the source nodes level, it updates the average output rates of every incoming stream. At the alpha network, it updates the output rates and window sizes of each alpha node. Then, it starts finding the cost of every join node, updating its window size and output rate at the same time. Finally, the monitor finds the cost of all joins and reports it to the optimiser.

The monitoring interval value can either be set by the user or decided by the system. Choosing a small interval increases monitoring overhead but provides a better chance for the optimiser to respond to quickly changing stream statistics. We can start with a small value and increase it if stream characteristics stay stable. Whenever it observes a change, the monitoring interval can be reduced again until the observed statistics stabilise.

6.5 Optimisation algorithm

The optimiser receives updates from the monitor about the current plan cost. While reacting to any change in the cost means a highly adaptive system, it has also some side effects (Babu and Bizzaro, 2005). Besides the increased overhead, this can lead to a case called thrashing. An Adaptive Query Processing system is thrashing if most of its resources are spent in adptivity-related overhead such as plan switching, not in query execution (Babu and Bizarro, 2005). Another reason to avoid adapting to every change is that this change might be transient, which means by the time the system switches into the new plan suitable for the change, the change is gone and the new plan is not suitable anymore. To avoid these problems, the optimiser keeps a window of plan costs and finds the average. If the new average cost significantly differs from the previous average, it starts the re-optimisation process.

The optimiser now starts trying to find a cheaper plan for the current conditions. Using dynamic programming techniques, we implemented two algorithms that use the cost model to find a new efficient plan.

6.5.1 Optimal plan algorithm

The first algorithm finds the optimal left-deep plan that is guaranteed to have a lower cost than all other possible left-deep plans based on the cost model. Working bottom-up, it starts by finding all possible plans of one-join size that join only two alpha nodes. It estimates every plan's cost and adds them to the first level table. In the second round, each one-join plan from the first level table is joined with every alpha node except those

participating in the current plan. This results in new, two-join plans that join three alpha nodes. Each new plan increments its cost with the new join's estimated cost; then, it is added to the second level table. This process continues until, after the final round – which is equal to the number of alpha nodes minus one – the last table contains plans that join all alpha nodes. The optimiser then simply chooses the cheapest plan from the table.

Procedure: Find optimal plan

Input: alpha nodes in the current plan $a_1, a_2, a_3, \ldots, a_n$

Output: New plan optP

```
1 rightParents = { a1, a2, a3, …, an}
2 leftParents = { a1, a2, a3, …, an}
3 for (r=0; r<n-1; r++)
4       for (i=0; i<leftParents.length; i++)
5               if (leftParents[i] ∈ rightParents)
6                       remove leftParents[i] from rightParents
7               if (leftParents[i] is a join node)
8                       rightParents = { a1, a2, a3, …, an}
9                       remove all alpha nodes in leftParents[i].plan from rightParents
10              for (j=0; j<rightParents.length; j++)
11                      if (leftParents[i] and rightParents[j] have a shared variable)
12                              jn = join (leftParents[i], rightParents[j])
13                              newPlan = leftParents[i].plan
14                              newPlan.add(jn)
15                              newPlan.cost += jn.cost
16                              table[r].add(newPlan)
17              leftParents = all last joins in table[r] plans
18 optP = cheapest plan in table[r]
19 return optP
```

Listing 6.1: Optimal plan algorithm

The pseudo code of the algorithm is presented in Listing 6.1. We have two lists representing potential left parents and right parents, initially containing all alpha nodes of the current rule. We then have three loops: the first represents the rounds, the second loops potential left parents, and the third loops potential right parents. In the first round (as a special case), where left parents are alpha nodes, lines 5 and 6 ensure that an alpha node does not get joined with itself. By deleting the current left parent alpha node from potential right parents, we prevent subsequent left parent alpha nodes from joining with the current

node, which results in a duplicate join node. For example, if we have three alpha nodes (A, B, and C) in the first round, we first hold A as the left parent. By deleting A from the right parents list, A is joined with B and, then, with C. In the left parents loop, B is the left parent, and the right parents list does not contain A; therefore, we do not join B with A (a duplicate to A join B). This is equivalent to having the second and third loops as:

```
for(i=0; i<leftParents.length; i++)
    for(j=i+1; j<rightParents.length; j++).
```

However, this logic does not hold for subsequent rounds, so we chose to remove nodes from the list of right parents. Lines 11–16 check if there is a possible join between the current left and right parents, create a join node, and add it to a new plan – which is initialised as the left parents' plan – and increment the new plan's cost with the new join's estimated cost. The new partial plan is stored in the current level table.

In the following rounds, the list of left parents will now be filled with join nodes (line 17). As the list of right parents is now empty, it is re-filled in line 8. Then, line 9 ensures that the left parent join node does not get joined with an already joined alpha node in the current plan, e.g. node (A&B) does not join A or B. The programme continues until the final round, where all plans join all alpha nodes. It chooses the cheapest plan and returns it.

To get an idea about the overhead of this algorithm, we estimate the number of join nodes it creates. We consider a star-joined plan, in which every alpha node shares a variable with each other alpha node. If we assume there are n alpha nodes in this plan, the first round will create (n-1)+(n-2)+(n-3)+…+1 = n(n-1)/2 partial plans of one-join size. Each will be joined with n-2 alpha nodes in the second round, resulting in n(n-1)/2 * (n-2) partial plans of two-join size. Each of these is joined with n-3 alpha nodes in the third round, i.e. there are n(n-1)/2 * (n-2)* (n-3) partial plans of three-join size. In the final round, each partial plan is joined with one remaining alpha node, so the number of plans is n(n-1)/2 * (n-2) * (n-3) * … * 1 = n(n-1)/2 * (n-2)!. This number serves as a worst-case scenario, as we assumed a star-shaped plan; other shapes result in a smaller number of alternative plans.

6.5.2 Greedy algorithm

The second algorithm uses a greedy approach to minimise the overhead. As in the first algorithm, it starts by finding all possible two-alpha joins and their costs. However, instead of saving and working on all of them in the next round, it only saves the cheapest one. This cheapest join then gets joined with all remaining alpha nodes, and the cheapest resulting

join is saved to get joined in the next round, and so on. Listing 6.2 shows a pseudo code of this algorithm. After joining the last alpha node, we get a new plan that is not guaranteed to be optimal, but the overhead can be significantly lower than the first algorithm.

```
Procedure: Find greedy plan
Input: alpha nodes in the current plan a1, a2, a3, …, an
Output: New plan grP
1 rightParents = { a1, a2, a3, …, an}
2 leftParents = { a1, a2, a3, …, an}
3 for (r=0; r<n-1; r++)
4       for (i=0; i<leftParents.length; i++)
5               if (leftParents[i] ∈ rightParents)
6                       remove leftParents[i] from rightParents
7               for (j=0; j<rightParents.length; j++)
8                       if (leftParents[i] and rightParents[j] have a shared variable)
9                               jn = join (leftParents[i], rightParents[j])
10                              estimate jn's cost and it to table[r]
11              jn = cheapest join in table[r]
12              remove jn.rightParent from rightParents
13              if (jn.leftParent is an alpha node)
14                      remove jn.leftParent from rightParents
15              clear leftParents
16              add jn to leftParents
17              add jn to grP
18              grP.cost += jn.cost
19 return grP
```

Listing 6.2: Greedy plan algorithm

While the number of join nodes created in the first round is the same as in the previous algorithm, it is significantly lower in the next rounds. In the second round, instead of joining every partial plan from the first round with the n-2 remaining alphas, only the cheapest partial plan is joined, resulting in n-2 partial plans of two-join size. Ultimately, there will be one complete plan and $n(n-1)/2 + (n-2) + (n-3) + \ldots + 1 = n(n-1)/2 + (n-1)(n-2)/2$ partial plans. This is also based on a star-shaped rule; other shapes require a smaller number of partial plans.

6.6 Plan migration

When the optimiser successfully finds a cheaper plan, it orders the rule engine to start a plan migration. Switching immediately from the old plan to the new plan will cause a loss

of possible results. This is because the new plan joins have empty memories, while the old joins' windows are full with intermediate results, which can be possibly joined by newly arriving tuples. To avoid this problem, we run both plans simultaneously until the new plan's joins are full with intermediate results and all the old plan join windows' tuples are expired. This period of time is equal to the largest window size of input streams participating in this plan. To avoid duplications in the output results, we only print the results of the old plan until the plan migration stage finishes; then, the new plan prints out its results, while the old plan is disconnected.

6.7 Conclusion

This chapter presented our second contribution: a cost-based adaptive optimiser for RDF streams. The optimisation process in R4 runs as follows: first, R4 produces an initial plan based on a number of known optimisation heuristics, such as plan sharing and avoiding Cartesian products. After running the initial plan, the monitor starts collecting statistics about the performance of operators, that are then used by the adaptive optimiser to find a cheaper plan. The adaptive optimiser estimates different plans costs based on a cost model. When a cheaper plan is found, the optimiser instructs the rule engine to start the migration process. In the next chapter, we validate the cost model, and evaluate the adaptive optimiser performance in different setting.

Chapter 7: Evaluating R4's Optimiser

Chapter 5 presented a comparative evaluation of the main reasoning engine. As the previous chapter addressed the optimisation problem in semantic stream processors, detailing the optimisation process carried out by R4, we now evaluate the performance of R4's optimiser. Before we evaluate the adaptive optimiser, which responds to changes in streams' characteristics by adapting the order of join nodes in its Rete networks, we investigate two known optimisation techniques: namely, minimising window sizes (Motwani et al., 2003) and sharing parts of the Rete networks between different rules (Forgy, 1982).

In Section 7.1, we examine the effect of minimising window size on the processing time and completeness of results. Then, Section 7.2 tests the ability of R4's optimiser to share operators between different rules and shows how this can improve the performance of the system. Finally, Section 7.3 evaluates the adaptive optimiser: first, by verifying its cost model and, second, by comparing the costs of the plans chosen by the adaptive optimiser with the plans chosen by the static optimiser in both stable and unstable streaming conditions. For all these experiments, we apply the same datasets used in the real-world scenario in the SemSorGrid4Env project described in Chapter 5, Section 5.1.

7.1 Quality of results vs. window size

In the information retrieval domain, in general, the measurements most commonly used to test the quality of results are precision and recall (Manning et al., 2008). The precision of a system is the proportion of retrieved material that is actually relevant. In contrast, the recall of a system is the proportion of relevant material that is actually retrieved (Van Rijsbergen, 1979). This pair is widely used to measure what is known as the effectiveness of the retrieval system. In other words, it is a 'measure of the ability of the system to retrieve relevant documents while, at the same time, holding back non-relevant one' (Manning et al., 2008).

Stream processing systems try to maximise both the precision and recall of the results they produce. However, it is not always possible to produce 100% correct and complete results when the systems are limited to a bounded amount of memory (Babcock et al., 2002). Approximation techniques used to handle the potentially unbounded input streams can affect both the precision and recall of the produced answers. The sampling technique

provides approximate answers that are not always correct, while the windowing technique provides incomplete but correct answers. The incompleteness used here regards the whole input stream, and is applied in situations when windowing is used as a mere resource management technique, i.e, not part of the intended semantics of the rule. In the second case, windows are used as temporal constraints, and results should be both correct and complete with regard to the applied window. As we do not use sampling techniques, we focus on examining the recall measurement[21].

To evaluate the quality of R4's results, we observed the completeness of results while applying different window sizes. We ran the system on two datasets – Dataset2 and Dataset3 – from Table 5.1. A rule that asks for all available swell period measurements observed by a sensor that has also observed a high wave (Listing 7.1) is applied. For every high wave observation, the system should search the windows for all swell period observations by this sensor. We stream the dataset, sending 169 observations every second (2366 triples) instead of every half an hour in the real-life streaming rate. In case of the second dataset, which has all half-hourly observations from a whole day, a 48 seconds window size would be equivalent to 24 hours window in the real-life streaming rate, and so should find all swell period values for every sensor with a high wave reading. Minimising windows caused the system to only find swell period readings that were observed near the high wave observation.

```
Forall ?s ?v ?v2(
If  And( ?ob [ssn:observedProperty -> waves:Wind_Wave_Height]
      ?ob [ssn:observedBy ?s]
      ?ob [ssn:observationResult ?result]
      ?result [ssn:hasValue ?value]
      ?value [ssnExt:hasQuantityValue ?v]
      External (pred:numeric-greater-than(?v 4.00))
      ?ob2 [ssn:observedProperty -> waves:Swell_Period]
      ?ob2 [ssn:observedBy -> ?s]
      ?ob2 [ssn:observationResult -> ?result2]
      ?result2 [ssn:hasValue -> ?value2]
      ?value2 [ssnExt:hasQuantityValue -> ?v2])
Then  ?s [ssnExt2:swellValue -> ?v2])
```

Listing 7.1: Completeness of results experiment rule

We first used Jena with the same rule and datasets to find the number of complete results for each dataset. To calculate the recall, we simply divided the number of retrieved results by the number of complete results.

[21] We compared R4's output to Jena's output and ensured that all retrieved results are correct.

Table 7.1 and Figure 7.1 show the increasing recall values in correlation with increasing window size. For each running, we also calculated the average response time. The table also shows the trade-off between the response time and completeness of results. While maximising window size means getting more results, it also means the response time will increase, as the system will spend more time processing the larger memories.

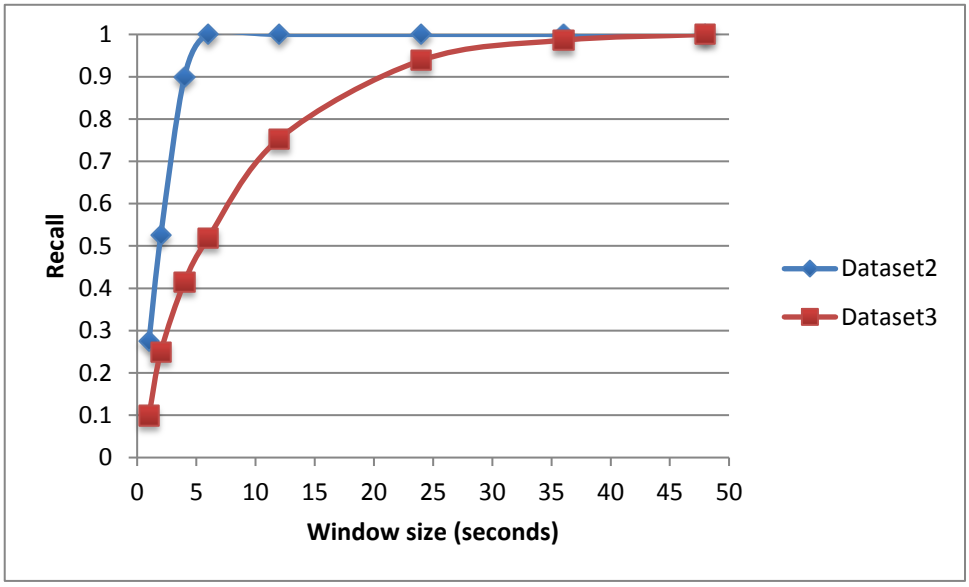


Figure 7.1: Window size vs. recall

Table 7.1: Recall and response time for different window sizes

Dataset	Window size (seconds)	No. of results	Recall	Average response time (ms)	Standard deviation
Dataset 2	1	11	0.28	2.52	0.23
	2	21	0.53	2.88	0.39
	4	36	0.90	3.04	0.56
	6	40	1.00	3.20	0.84
	12	40	1.00	3.20	0.66
Dataset 3	1	65	0.10	2.92	0.80
	2	163	0.25	3.88	0.77
	4	271	0.42	4.67	0.80
	6	339	0.52	5.91	1.01
	12	492	0.75	6.22	1.40
	24	613	0.94	7.99	1.08
	36	644	0.99	8.47	1.24
	48	653	1.00	6.65	1.16

7.2 Operator Sharing

```
Forall ?ob ?v ?s (
  If  And( ?ob [ssn:observedProperty -> waves:Wind_Wave_Height]
       ?ob [ssn:observedBy -> ?s]
       ?ob [ssn:observationResult -> ?result]
       ?result [ssn:hasValue -> ?value]
       ?value [ssnExt:hasQuantityValue -> ?v]
       External (pred:numeric-greater-than(?v 2.00)))
  Then  ?s [ssnExt2:highAlert -> ?v])
Forall ?ob ?v ?avg(
  If  And( ?ob [ssn:observedProperty -> waves:Wind_Wave_Height]
       ?ob [ssn:observedBy -> ?s]
       ?ob [ssn:observationResult -> ?result]
       ?result [ssn:hasValue -> ?value]
       ?value [ssnExt:hasQuantityValue -> ?v]
       (?avg = External (func:numeric-avg(?v)) ) 30 m
  Then  ?s [ssnExt2:average -> ?avg])
```

Listing 7.2: Operator sharing experiment rule

R4 enables sharing of operators and their memories between multiple rules. To examine how operator sharing can affect the memory consumption of the system, we created two rules that can share some of their operators. The two rules are then executed in two settings: they share their operators in the first setting, while they run separately in the second. For both settings, we periodically measure the memory size of all stateful operators. Then, we compare the shared network memory consumption to the unshared one.

The implemented rules are presented in the RIF in Listing 7.2. The first rule finds wave heights greater than a specified threshold, along with the sensor ID that observed it. The second rule finds the average wave height across all 24 sensors every half hour. The unshared Rete networks are illustrated in Figure 7.2, while the shared network is depicted in Figure 7.3. It shows that, for this particular use case, most filter operators are shared, and three out of four joins are also shared.

We ran those rules on Dataset3, which contains nearly 100k triples. We set the global window size to 8 seconds (equivalent to 4 hours, as we run in the same input rate as the previous experiment). All alpha and beta memories report their sizes every second. We calculate the total size of these memories every second in both settings and compare them to show the effect of operator sharing on memory consumption. Full numerical results can be found in Appendix C. Here, we represent two charts: one shows only the totals of

memory sizes for both settings, and the other shows the growth of all window sizes over time.

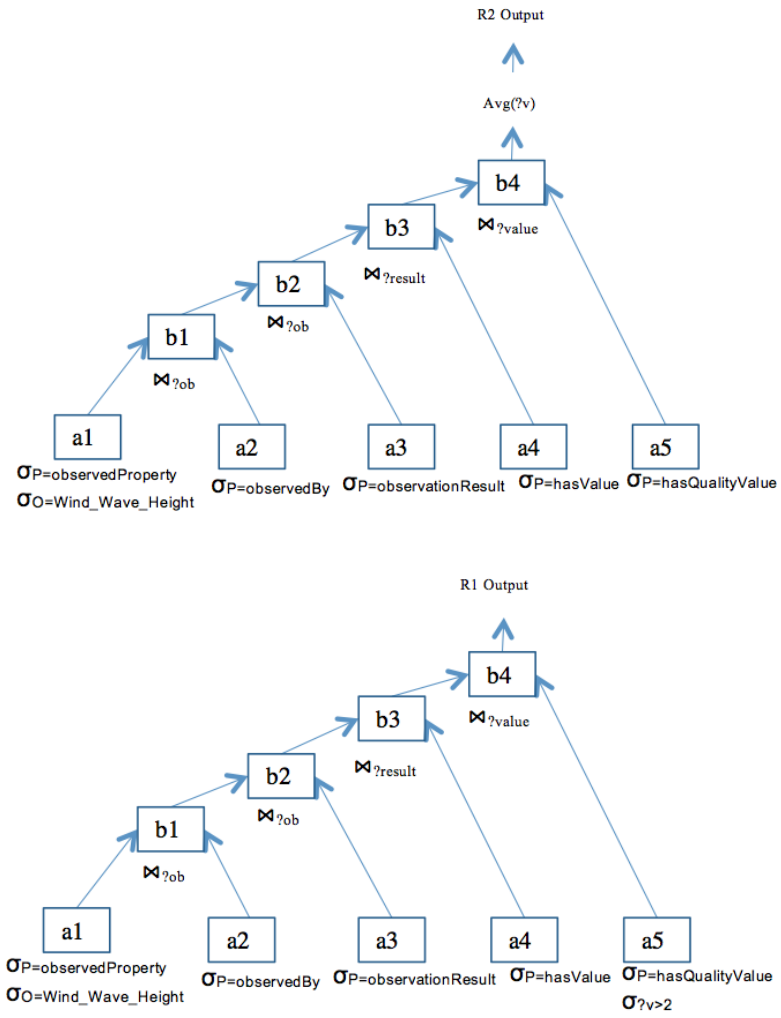


Figure 7.2: Two separate Rete networks for two rules without sharing

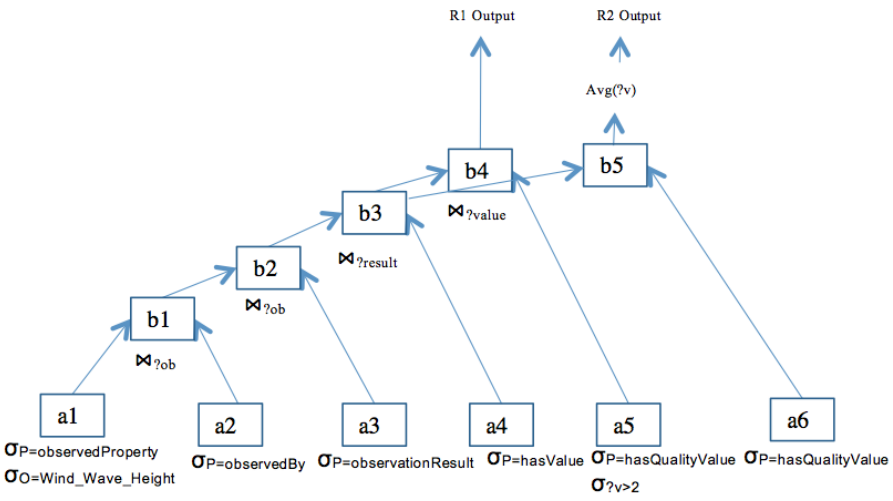


Figure 7.3: Shared Rete network for two rules

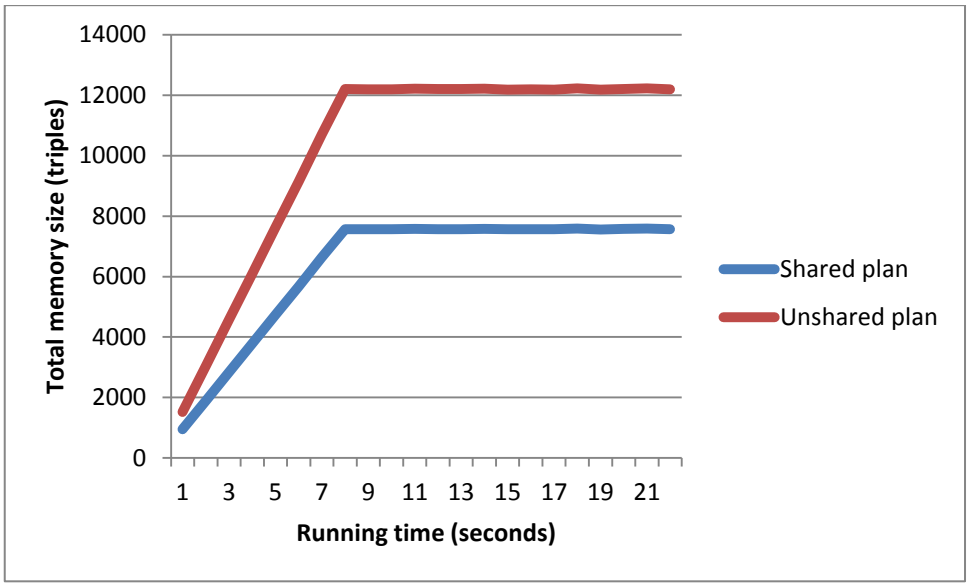


Figure 7.4: Total memory sizes for shared and unshared plans

Figure 7.4 simply shows that the shared plan consumes less memory than the unshared one. The extra memory saved by the shared plan can be exploited by enlarging the shared windows in order to maximise coverage and increase the completeness of the results.

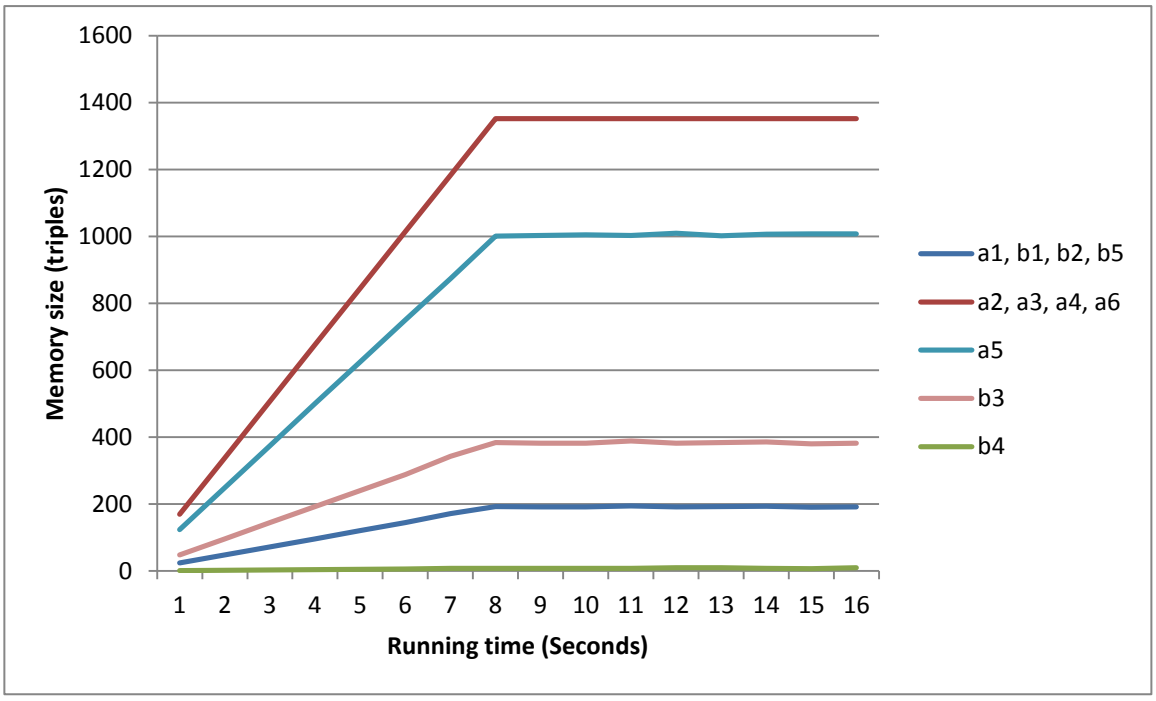


Figure 7.5: Individual memories growth over time

Figure 7.5 shows the behaviour of the windows in the shared plan over time. The windows all start from zero and grow incrementally until they reach their predefined limit (8 seconds), at which point they start dropping expired tuples while receiving new tuples at the same time. The pace at which they evolve, however, is different. The figure shows that the highest memory size belongs to alpha memories with only one condition matching the predicate (a2, a3, a4, and a6). In this use case, these predicates are not selective, as there is one triple matching each predicate in every observation. The first alpha memory (a1) is connected to a more selective filter, its selectivity determine the selectivities of its following join nodes (b1, b2, b5). The lowest memory size is associated with the b4, which is beta memory of the last join in the first rule, connecting five triple patterns, in which two of them are selective. Alpha and beta memories of the unshared plan follow the same patterns, but as some nodes and their memories are duplicated, we get the higher memory cost.

7.3 Evaluating the adaptive optimiser

R4 has an adaptive optimiser that responds to changing streams' characteristics during runtime by adapting the order of join nodes in the running networks. The adaptive optimiser communicates with the monitor to receive real-time statistics about the cost of the running plan, tries to find a cheaper plan based on a cost model, and instructs the rule engine to migrate to the new plan if found.

Before evaluating the performance of the adaptive optimiser, we need to validate the employed cost model. Experiments in Section 7.3.1 test the optimiser's ability to correctly estimate the performance of different plans by comparing measured plans' costs and estimations based on the cost model. Section 7.3.2 evaluates the decisions taken by the adaptive optimiser at stable conditions. We compare the cost of the plan chosen by the adaptive optimiser (based on the cost model) to the cost of the initial plan chosen by the static optimiser. Finally, Section 7.3.3 repeats the previous experiment but under unstable conditions, where stream conditions change during the lifetime of the running rule.

7.3.1 Verifying the cost model

The cost-based optimisation process in R4 uses real-time statistics about the input streams to measure the cost of the running plan and estimate costs of possible alternative plans using the cost model, presented in Section 6.3. The cost model is a vital part of the

optimisation process, as it can lead to wrong decisions if it does not estimate costs of different plans correctly. While it is not expected to provide accurate measures of the costs, it should be able to correctly rank different plans based on their costs, especially when their costs vary significantly. This section presents experiments that compare the estimated costs of different plans to their actual costs.

7.3.1.1 Methodology

To verify the cost model, we run two experiments for two rules: simple and complex. The simple one (triple patterns are shown in Listing 7.3) takes input from a single stream and has intra-graph joins only, while the complex rule (Listing 7.4) takes input from multiple streams and has both intra- and inter-graph joins. For each experiment, we compare the measured and estimated costs of all feasible plans. Feasible plans are plans that follow the shared variable condition employed by the static optimiser.

Both measured and estimated costs are calculated using the cost model. However, the measured cost is based on the actual run-time statistics of input and output rates, beta memories' sizes, and operators' selectivities. On the other hand, the estimated cost is based on estimations of operators' output rates, beta memories' sizes, and operators' selectivites. To further verify the cost model in the first experiment, we compare the measured plan cost – based on the cost model – to the actual performance of the plan in terms of its response time, i.e., the time between receiving new elements till producing the last output of each update.

The first experiment's rule represents a basic pattern matching with three conditions (or triple patterns) that checks if an observation's result is above a specified threshold. For this rule, there are only two feasible plans: (A⋈B)⋈C, and (B⋈C)⋈A, as plan (A⋈C)⋈B violates the shared variable condition. For clarity, we omit parentheses and the join symbol in the following and use the conditions' ordering to represent plans, e.g., (A⋈B)⋈C is represented as ABC and (B⋈C)⋈A as BCA. For each plan, we print latency, measured cost, and estimated cost every second while inserting a stream of 700 graphs (9800 triples) per second input rate. We use a 1.1 million triples input file that contains observations collected over 10 days. We manipulated the observation results so that they only match the rule condition (greater than 100) after 70 seconds to check the cost model's sensitivity to the changing selectivity.

```
A    ?ob ssn:observationResult ?result
B    ?result ssn:hasValue ?value
C    ?value ssnExt:hasQuantityValue (?v>100)
```

Listing 7.3: Triple patterns of Rule 1

In the second experiment, the rule checks if there are two sensors observing the same high result. In total, there are 16 different feasible plans to implement this rule. We chose five of them that are expected to have different costs, including: ABCFED, DEFCBA, CBAFED, FEDCBA, and FCBADE. We run each plan individually, printing measured and estimated costs every second. As the plan receives input from two streams, we obtained observations of two sensors and stored them in two files to stream them at a faster rate. In this experiment, the first stream input rate is 2000 triples/second, and the other stream's input rate is 1000 triples/second, while the CCO sensors provide data at an output rate of 98 triples (seven graphs or observations) per 30 minutes. The content of the streams is not manipulated in this experiment, and both streams are assigned global window sizes of five seconds.

```
A    ?ob1 ssn:observationResult ?result1
B    ?result1 ssn:hasValue ?value1
C    ?value1 ssnExt:hasQuantityValue (?v>100)
D    ?ob2 ssn:observationResult ?result2
E    ?result2 ssn:hasValue ?value2
F    ?value2 ssnExt:hasQuantityValue (?v>100)
```

Listing 7.4: Triple patterns of Rule 2

7.3.1.2 Results

For the first experiment, Figure 7.6 compares the measured and estimated costs for each of the two alternative plans plotted every second during the lifetime of the rule. The figure shows that the second plan (BCA) outperforms the first one in the first 70 seconds and has the same cost as the first plan after that. This is because triple pattern C is very selective at the beginning, while the selectivities of patterns A and B are equal to one, i.e. every input graph has one triple matching A and one triple matching B. This means that joining pattern C with B first results in small beta memories for the whole network. On the other hand, joining pattern A with B first results in a join node that has an output rate equal to the original stream input rate (700 matches/second), leading to a big beta memory following this join. As we manipulated the input stream so that every graph after second 70 matches

triple pattern C, all joins now have the same selectivity, which means that both plans have the same cost.

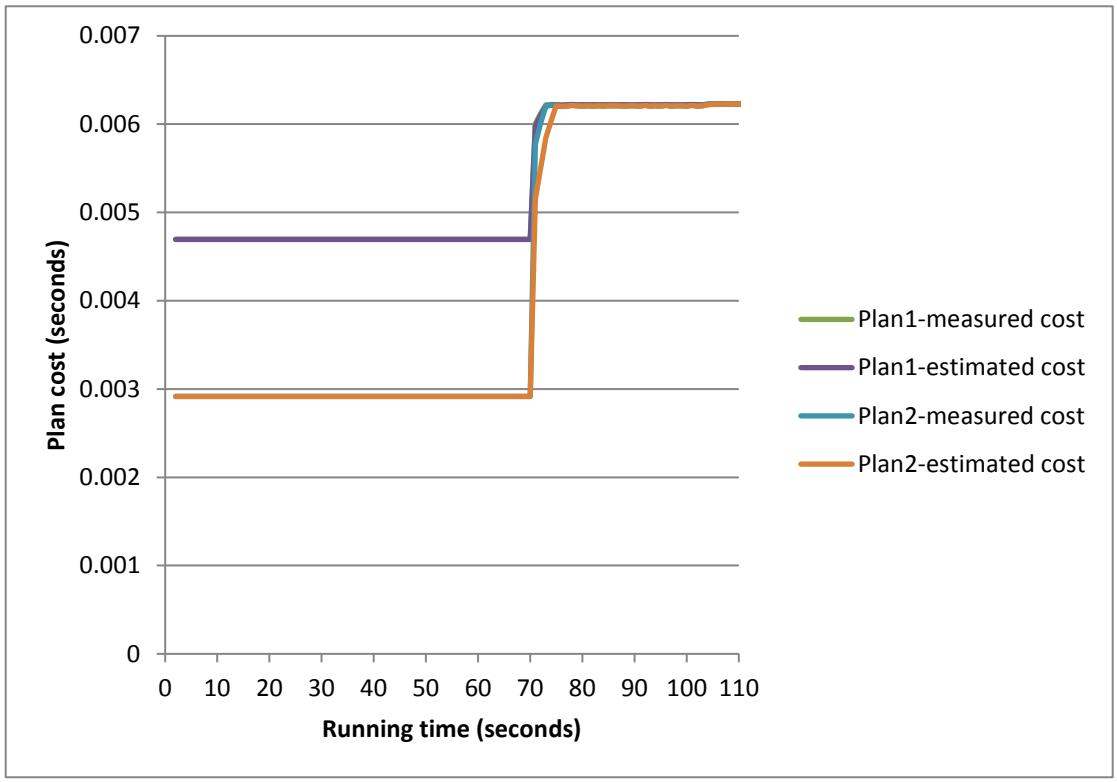


Figure 7.6: Estimated and measured costs for Rule 1 plans

In this setting, the figure also shows that the measured and estimated costs of both plans are identical. While it is not usually the case, the cost model can accurately calculate the cost of simple plans, where only intra-graph joins are used. The input observations (graphs) represent perfect characteristic sets. In this case, the output rate of the whole plan is based on the output rate of the most selective pattern (pattern C in this case). As we get actual statistics from the alpha network, selectivities and output rates of the beta network nodes, and therefore, the cost, can be accurately calculated.

Matching measured and estimated costs proves that the optimiser performs well in terms of estimating different parameters that are used in calculating the estimated cost based on the cost model, including output rates, beta memories' sizes, and operators' selectivities. Furthermore, in this setting, we also measure the actual plan latency and compare it to the measured cost to see how well the cost model formulas reflect the real-life cost. Figure 7.7 shows that, while the plans' costs produced by the cost model underestimate the actual time taken to process the input, they reflected the same trend. Before 70 seconds, both show that Plan 2 outperforms Plan 1. At 70 seconds, the costs of both plans increase – Plan 1 slightly and Plan 2 significantly. After this turning point, both methods show that both

plans perform equally. The underestimation could be due to the constant costs of handling one triple in the cost model formulas being underestimated in addition to the fact that the cost model only calculates the costs of join nodes; other costs such as filtering at the alpha network, generating results at the terminal node, and communication costs between nodes are not included.

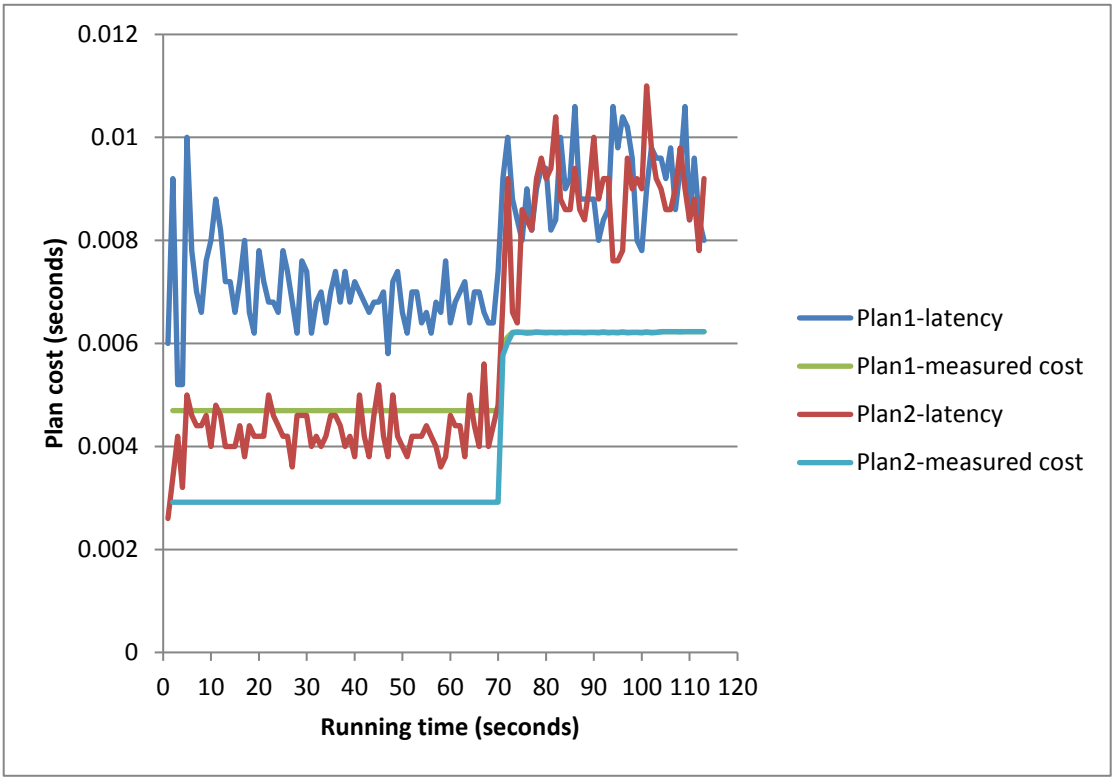


Figure 7.7: Measured costs vs. latency for Rule 1 plans

The second experiment also compares measured and estimated costs but for a more complex rule. The chosen plans are as follows: Plan 1 (ABCFED) matches stream 1 patterns (with the most selective pattern – pattern C – at the end) with stream 2 patterns (the most selective pattern F has to precede other patterns, as it is the only one that shares a variable with stream 1 patterns). Plan 2 (DEFCBA) matches stream 2 patterns (with the most selective pattern – pattern F – at the end) with stream 1 patterns. Plans 3 and 4 (CBAFED and FEDCBA) are similar to the previous two but with the most selective patterns for each stream first. The last plan (FCBADE) is different, as it performs the inter-graph join first. As expected, Figure 7.8 (with measured costs), shows that plans 3 and 4 outperform plans 1 and 2, as they have the more selective patterns first. Between them, plans with stream 2 patterns first outperform their equivalent plans where stream 1 patterns

are placed first. This is because stream 2 has a lower input rate than stream 1. Plan 5 performs between Plans 2 and 3.

More importantly, this ranking is preserved in Figure 7.9, which shows estimated costs of the five plans[22]. While the measured and estimated costs do not exactly match, the cost model was accurate enough to rank different plans correctly.

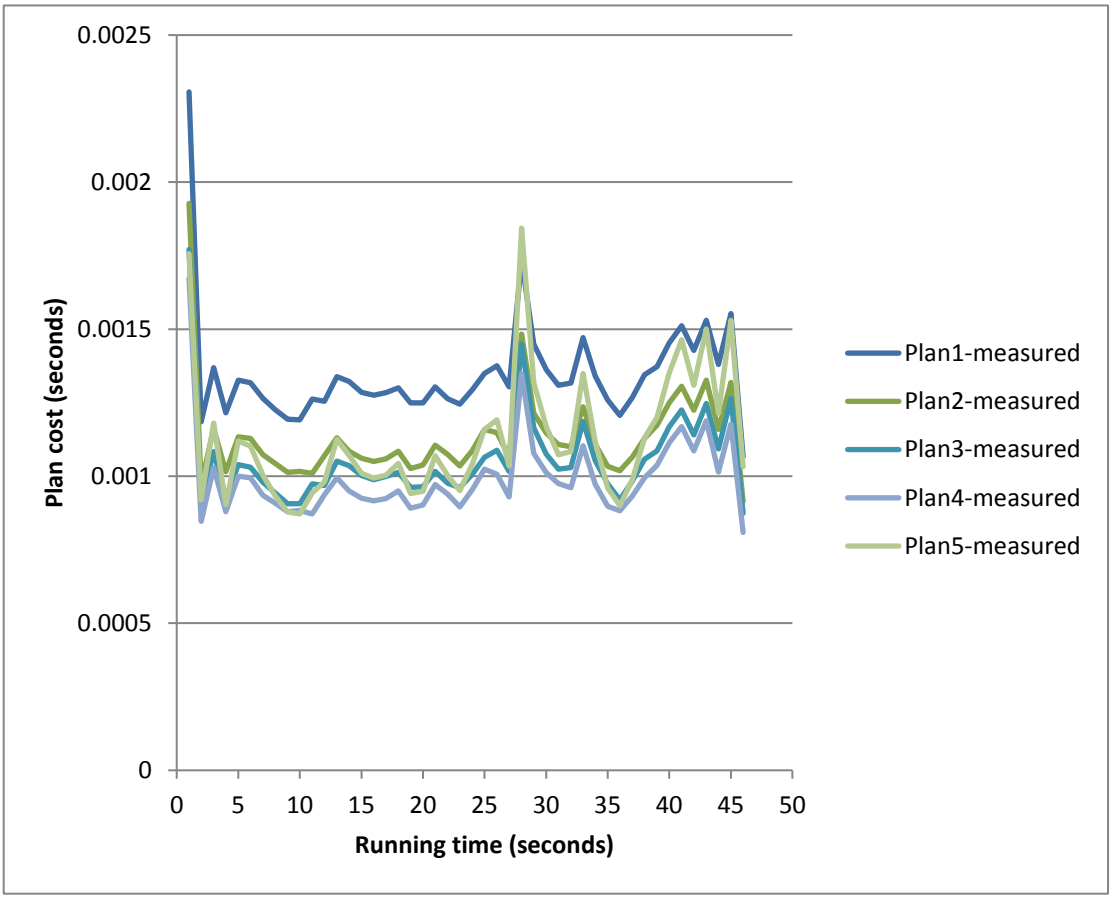


Figure 7.8: Measured costs for Rule 2 plans

7.3.2 Optimisation performance under stable conditions

After we ensured that the cost model could reasonably estimate different plan costs, we now measure the performance gains after employing the cost-based adaptive optimiser. The absence of streams' statistics beforehand means that the initial plan chosen by the static optimiser can be inefficient. However, as soon as the monitor collects some statistics, the adaptive optimiser uses them to find and switch to a more efficient plan. In this section,

[22] We split the results into two figures, as they do not look clear in a single graph; raw results and a single graph for measured and estimated costs can be found in Appendix C.

we compare the measured costs of the initial plan and the new plan chosen by the adaptive optimiser when stream conditions – input rates and selectivities – are stable.

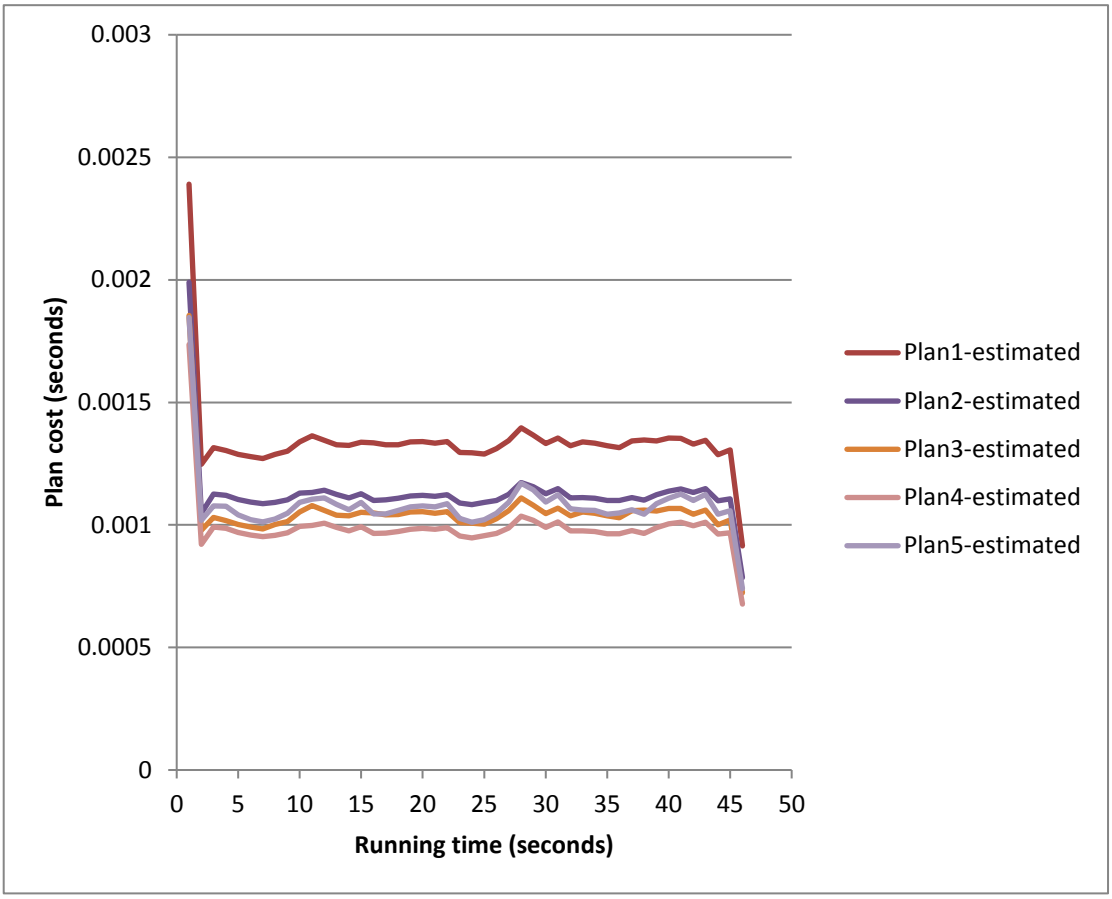


Figure 7.9: Estimated costs for Rule 2 plans

7.3.2.1 Methodology

We apply the same two – simple and complex – rules from the previous section (Listings 7.3 and 7.4). For each, we first run the static optimiser only, identify the chosen plan, ask the monitor to print the measured cost of the running plan every second until the end of the streamed files, and find the average cost. We apply the same process using the adaptive optimiser twice, first using the greedy algorithm and then using the optimal algorithm. We only show the cost of the plan chosen by the optimal algorithm if it is different to the one chosen by the greedy algorithm. Moreover, to show the overhead of adaptivity, we run the adaptive optimiser (using the greedy algorithm) in two settings: the first one switches to the new chosen plan immediately without employing the migration process (during which both old and new rules are running), and the second applies the plan migration process.

For each rule, we compare the costs of plans chosen by the static and adaptive optimisers using a variety of parameters that affect plan costs. These include using different global

window sizes for the input streams, different stream input rates, different operators' selectivities, and different numbers of join nodes.

7.3.2.2 Results

Varying window sizes:

For the first rule, Figure 7.10 compares different plans' costs using window sizes varying from 1–15 seconds. The selectivity of operators reflects the real-life sensor data, as we have not manipulated the content of streams, while the input rate is fixed at 70 graphs/second (almost 1000 triples/second). The initial plan generated by the static optimiser follows the same order the rule was written in (i.e. ABC), while both the greedy and optimal algorithms of the adaptive optimiser chose the order CBA, putting the most selective pattern first. For all the different window sizes used, the adaptive plan's cost is 44% less than the initial plan.

We also notice that increasing the window size only slightly increases the plan cost for both static and adaptive plans. A bigger window size means elements will stay longer in memories, affecting mainly the probe cost (and possibly the result generation cost, as there is a bigger chance of matches but not the insertion or deletion costs). Thanks to the hash-based implementation of alpha and beta memories, bigger windows do not dramatically increase plan costs.

However, increasing window sizes has another side affect, which is a more expensive plan migration. The figure shows this clearly; as for the first window size, the plan migration stage only caused less than a 2% increase in the adaptive plan cost, while the last window (15 seconds) caused almost a 20% increase. This behaviour is expected, as our implemented plan migration strategy runs both the old and new plans for an amount of time equal to the window size in order to ensure complete results (with regard to the requested window size). The plans' costs presented are averaged over a period of 100 seconds. Running the same rule for longer time in the same stable conditions reduces the cost of migration relevant to the static plan, i.e. the static plan cost will remain the same, while the high cost of migration will be averaged over a longer period, resulting in a lower cost.

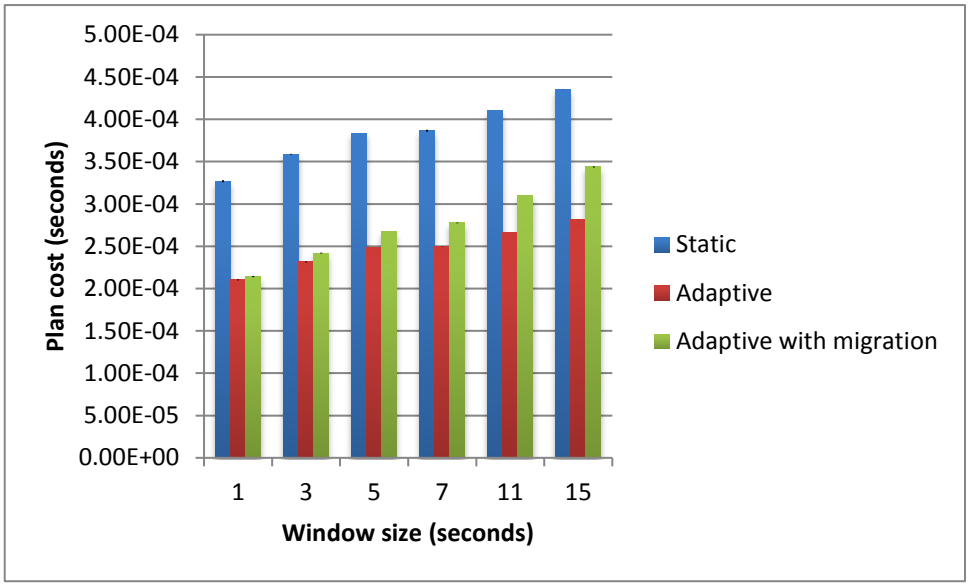


Figure 7.10: Average plans' costs using different window sizes for Rule 1

For the second rule, we chose window sizes of one and five seconds, trying their different combinations for the two streams, as shown in Figure 7.11. Both streams have the same input rate of 1000 triples/second. The static plan puts the first stream triple patterns with the same order as they appear in the rule (ABC) and then the second stream patterns in reversed order (the most selective pattern first: FED) to follow the shared variable condition. On the other hand, the greedy algorithm finds that joining the most selective pattern of the two streams (C and F) is the cheapest among two-alpha-nodes networks. It then joins the remaining patterns of the first stream (A and B) and then the patterns of the second stream (D and E) for the first, second, and last setting, producing the plan CFBAED. For the third setting, as the window of the first stream is bigger, it joins the remaining patterns of the second stream before those of the first stream, producing the plan CFEDBA. The optimal algorithm chooses a different order: CBFEAD for the first, second, and last settings and FECBDA for the third setting.

We notice that the optimal plans only marginally outperform the plans generated by the greedy algorithm. Gains over static plans are between 15% to 25%, which is less than in the previous experiment. The reason could be that the second half of this rule is already optimised in the initial plan, placing the most selective pattern first in order to follow the shared variable static optimisation technique. Finally, we notice that plan migration costs are similar in the last three settings, as the adaptive optimiser has to run both old and new plans until the bigger window of both streams finishes.

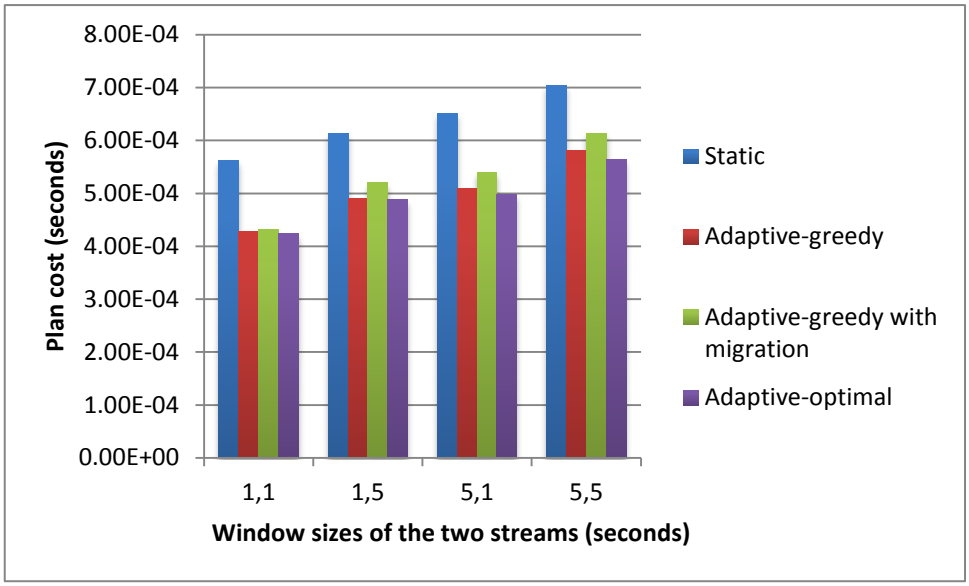


Figure 7.11: Average plans' costs using different window sizes for Rule 2

Varying input rates:

In this experiment, we fix window sizes at five seconds and vary the streams' input rates. Figures 7.12 and 7.13 compare costs of static and adaptive plans for different input rates for Rules 1 and 2. The same observations made in the previous experiment are noticed here, including that both greedy and optimal algorithms pick the same plan for the simple rule and different plans for the second rule but with marginal cost differences, and that gains over the static plan are higher in Rule 1. However, an important difference that can be noticed is the steep increase of plan costs with a higher input rate. Unlike increasing window sizes that only affect the probing cost, increasing input rates affect all join operations. Higher rates mean more insertions, leading to more deletions when they expire, and also to more probing operations.

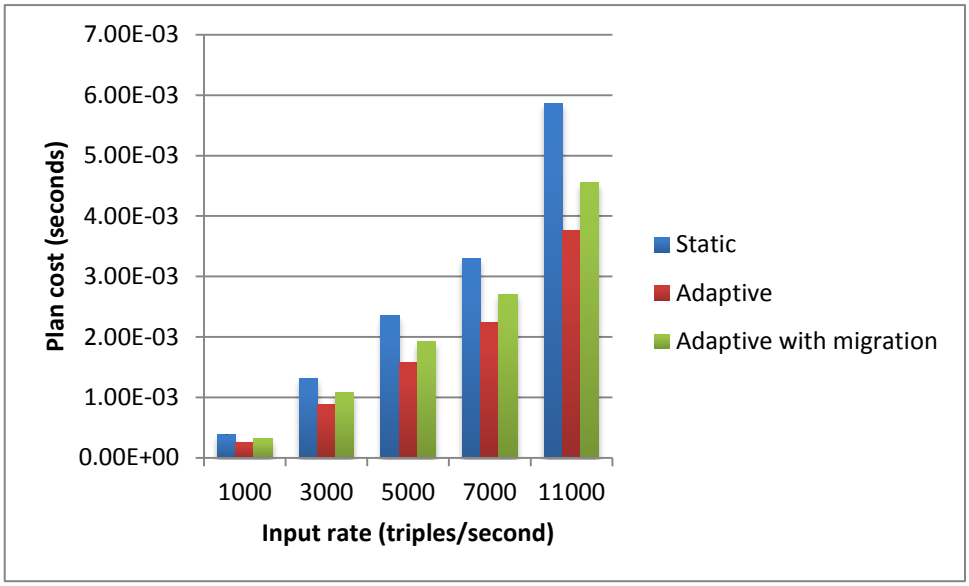


Figure 7.12: Average plans' costs using different input rates for Rule 1

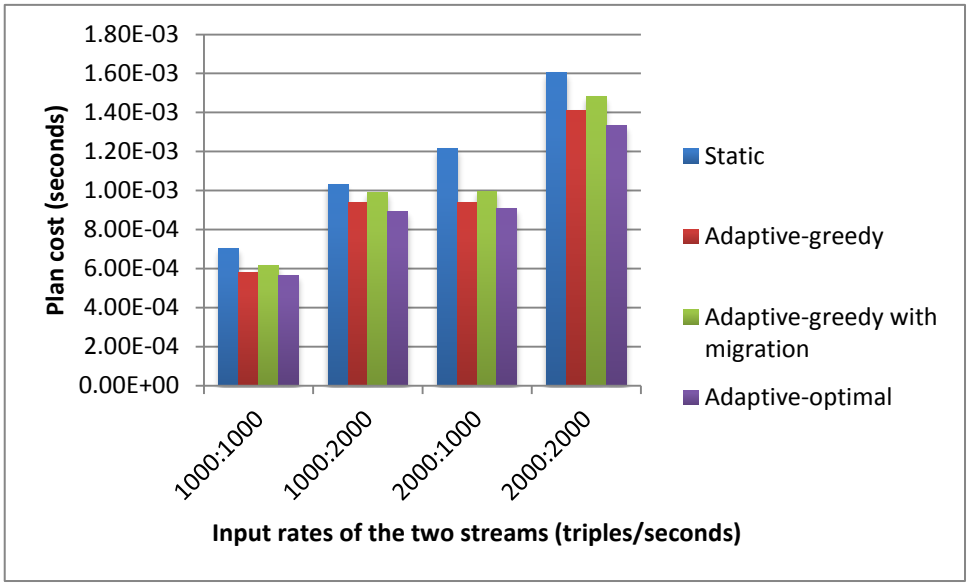


Figure 7.13: Average plans' costs using different input rates for Rule 2

Varying selectivities:

Operators' selectivities are based on the values contained in the streamed data. Selectivities of operators are measured using the selectivity factor, which takes values between 0 and 1. A low selectivity factor near 0 means that the operator is highly selective, producing only a small fraction of its input as output. On the other hand, a high selectivity factor near 1 means that the operator produces most of its input as output, indicating low selectivity.

In the previous two experiments, we controlled window sizes and input rates but have not manipulated the actual dataset, so they reflect real-world observation values. In this experiment, we fix window sizes and input rates, while varying the input data values to

result in different selectivities. We prepared several datasets, changing observation results (triples matching triple pattern C) in each of them to match certain selectivity. For example, half of the observations in the '0.5' dataset match the rule conditions, and all of the observations in the '1' dataset will make it to the output.

Figure 7.14 shows that the best gains are obtained when the selectivity is low. When the selectivity goes up to 0.75, the cheaper plan found by the adaptive optimiser (CBA) becomes more expensive than the static plan (ABC) with the migration costs. This situation can be avoided by applying a threshold before changing to any cheaper plan e.g., changing plans only if the new plan is 25% cheaper than the current plan. When the selectivity is 1, the adaptive optimiser finds that the cost of the new plan is the same as the cost of the old one (as now all triple patterns have the same selectivity), and therefore, it does not change plans.

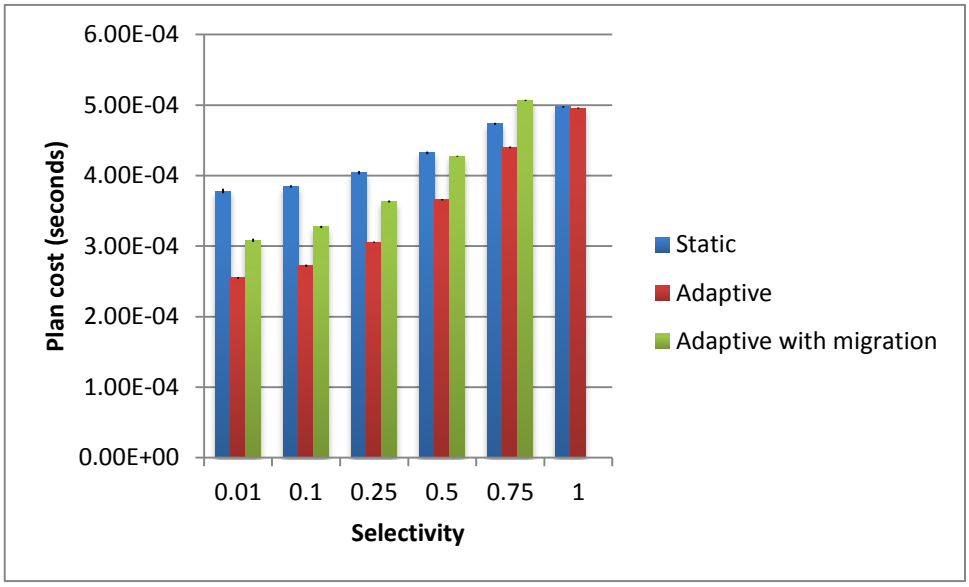


Figure 7.14: Average plans' costs using different selectivities for Rule 1

For the second setting, we join the stream of '0.1' selectivity with the streams of '0.01', '0.05', and '0.1' selectivities. Results are similar to the previous setting, getting better gains with lower selectivies. However, in this case, the greedy algorithm failed to find a cheaper plan. It chooses to join patterns C and F first (as described in varying window sizes experiment), which ends up in a plan that is more expensive than the static plan.

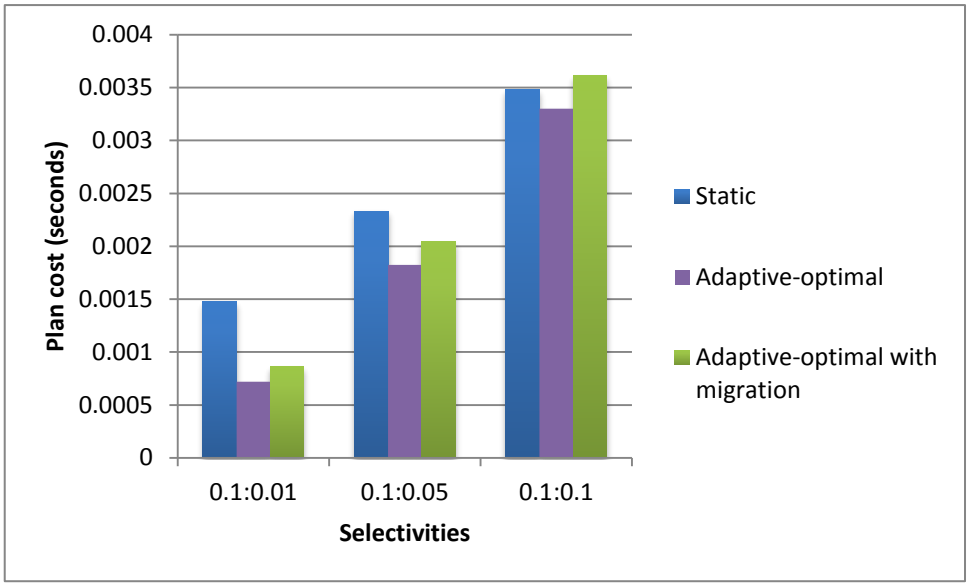


Figure 7.15: Average plans' costs using different selectivities for Rule 2

```
G    ?ob rdf:type ssn:Observation
H    ?ob ssn:featureOfInterest :PhysicalMetOcean
I    ?ob ssn:observedProperty :Mean_Wave_Direction
J    ?ob ssn:observedBy ?sensor
A    ?ob ssn:observationResult ?result
B    ?result ssn:hasValue ?value
C    ?value ssnExt:hasQuantityValue (?v>100)
```

Listing 7.5: Triple patterns added to Rule 1

Varying number of joins:

In this experiment, we check how the adaptive optimiser performance scales while increasing the number of joins. For the first setting, we add one more triple pattern to Rule 1 each time, causing one more join, until we reach the rule of seven triple patterns presented in Listing 7.5.

Figure 7.16 shows that adding triple patterns G and H to the front of the rule causes a big increase in the static plan cost, as these patterns are not selective (every graph in the dataset satisfies them). The adaptive plan cost also increases to a lesser extent, as these patterns are placed at the end of the network. Adding the third triple pattern (I) is beneficial to the static plan, as it is a selective pattern. The adaptive plan cost is not changed, as its first join of C and B is already more selective than this.

```
A    ?ob1 ssn:observationResult ?result1
B    ?result1 ssn:hasValue ?value1
C    ?value1 ssnExt:hasQuantityValue (?v>100)
D    ?ob2 ssn:observationResult ?result2
E    ?result2 ssn:hasValue ?value2
F    ?value2 ssnExt:hasQuantityValue (?v>100)
K    ?ob3 ssn:observationResult ?result3
L    ?result3 ssn:hasValue ?value3
M    ?value3 ssnExt:hasQuantityValue (?v>100)
```

Listing 7.6: Rule 2 after joining one more stream

With Rule 2, instead of adding more triple patterns to two parts of the rule concerning the two input streams, we add more streams, asking for the matching observation results across three and four streams. The original Rule 2, which matches two streams with five joins, adds two more streams, increasing the number of joins to eight and 11. Similar to the previous experiments' varying window sizes and input rates, the optimal algorithm produces plans that only slightly outperform the greedy plans for five and eight joins. Having 11 joins, however, causes the current implementation of the optimal algorithm to crash (out of memory exception). The total number of possible plans (disregarding the shared variable condition) is equal to 10!, i.e. 3,628,800 plans. Even for more efficient implementations of the algorithm, we expect that it would take too long to find a good plan – that stream conditions might change again before applying the new plan, rendering it useless. For big beta networks in a highly fluctuating stream environment, the fast, suboptimal greedy algorithm would be preferable.

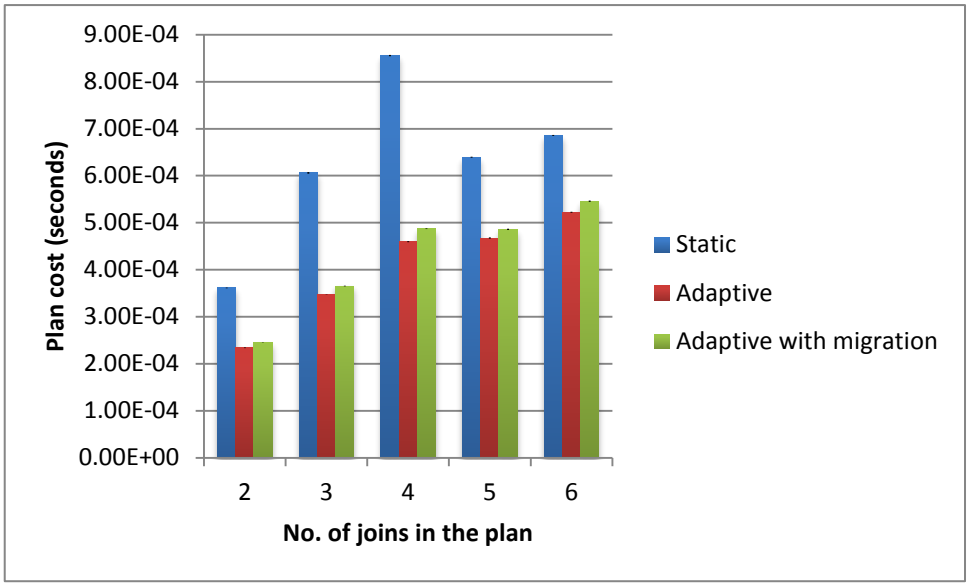


Figure 7.16: Average plan's cost for an increasing number of joins in Rule 1

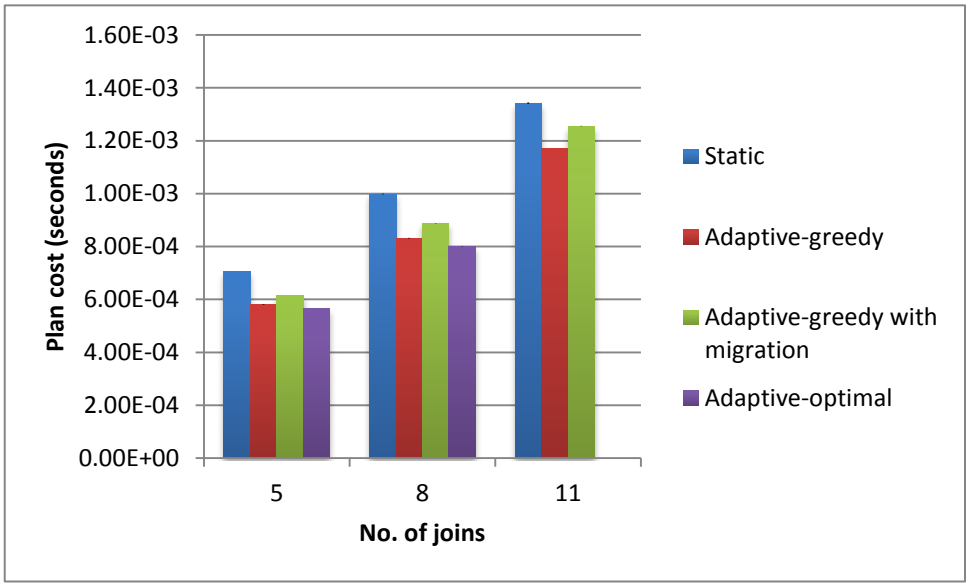


Figure 7.17: Average plan's cost for an increasing number of joins for Rule 2

7.3.3 Optimisation performance under unstable conditions

The experiments in the previous section showed that the adaptive optimiser in almost all cases was able to produce a cheaper plan. However, the gain over the static plan depends mainly on the rule itself and if there is any room for improvement. For example, if the simple rule (Rule 1) was originally written with the most selective pattern first, the adaptive optimiser would not be able to provide any improvements. Yet, the adaptive optimiser's main goal is not only to provide a better plan than the initial one but to adapt to a changing environment. In this section, we evaluate the adaptivity of the optimiser by

testing it under an unstable environment, where stream conditions change during the lifetime of the rule.

7.3.3.1 Methodology

We observe the cost of the static and adaptive plans over runtime while varying stream conditions. Out of the four parameters that affect plan costs tested in the previous section, only input rates and selectivities are related to streams; window sizes and number of joins are related to rules decided by the user, and therefore, are not expected to change rapidly during runtime. Therefore, we conduct two experiments measuring how the optimiser responds to changes in stream input rates and changing selectivities. For both experiments, we use the rule presented in Listing 7.6 that joins three streams, checking if they all observe the same value. All streams in both experiments are observed through a window size of three seconds.

In the first experiment, we run the rule for 60 seconds. During the first 20 seconds, we set the first stream to a high input rate of 3000 triples/second, while the second and third streams have medium, 300 triples/second, and low, 30 triples/second, input rates, respectively. At 20 seconds, we swap the input rates of the first and third streams so that the first has the low rate and the last has the high rate while the second remains the same. After another 20 seconds, we swap the rates of the first and the third streams again so that they return to their original rates.

For the second experiment, a similar setting is used. We prepare three input streams with manipulated observation values to reflect the following selectivities: high (0.1), medium (0.05), and low (0.01) selectivity factors for the first, second, and third streams, respectively. After 45 seconds, data in the three streams begin to reflect different selctivities, in which the first stream becomes highly selective (low selectivity factor 0.01) and the third stream data represent lower selectivities (high selectivity factor 0.1). At 90 seconds, they go back to reflect their original selectivities.

7.3.3.2 Results

Figures 7.18 and 7.19 show the effect of changing input rates and selectivities on the costs of the static and adaptive plans. A similar pattern is observed in both figures[23], showing

[23] We note that Figure 7.19 provides more neat patterns, because selectivities, input rates, window sizes are all controlled, while selectivities in Figure 7.18 reflect unmanipulated real-world data.

that the adaptive optimiser immediately responds to the changing conditions by switching to a more efficient plan, maintaining a low cost most of the running time. The static plan cost, on the other hand, rises and falls with the changing conditions without any control.

During the first part of the experiment, the static plan follows the order of the original rule, joining the first stream with the second and, then, the third. This results in a high cost, as the first stream has a higher input rate in the first experiment and a lower selectivity in the second one. The adaptive optimiser chooses to join the third stream with the second and first stream, producing a lower cost plan. When the first change occurs, the cost of the static plan drops down, as the first stream now has the lowest input rate (or highest selectivity in the second setting). The adaptive optimiser immediately notices that its current plan (that joins the third stream first) becomes inefficient and successfully changes to a plan that joins the first stream first (as in the static plan). However, it goes through a short period of high costs during plan migration, as it has to run both plans simultaneously. After the second change, the static plan goes into its original high cost, while the adaptive optimiser successfully changes plans to join the third stream first, maintaining its low cost after the end of migration.

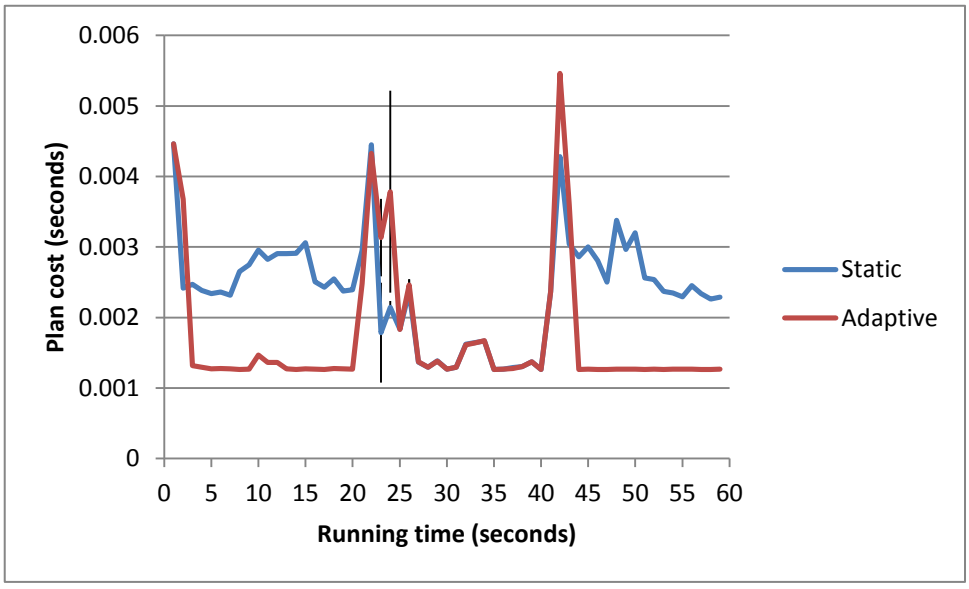


Figure 7.18: Adaptivity while changing input rates

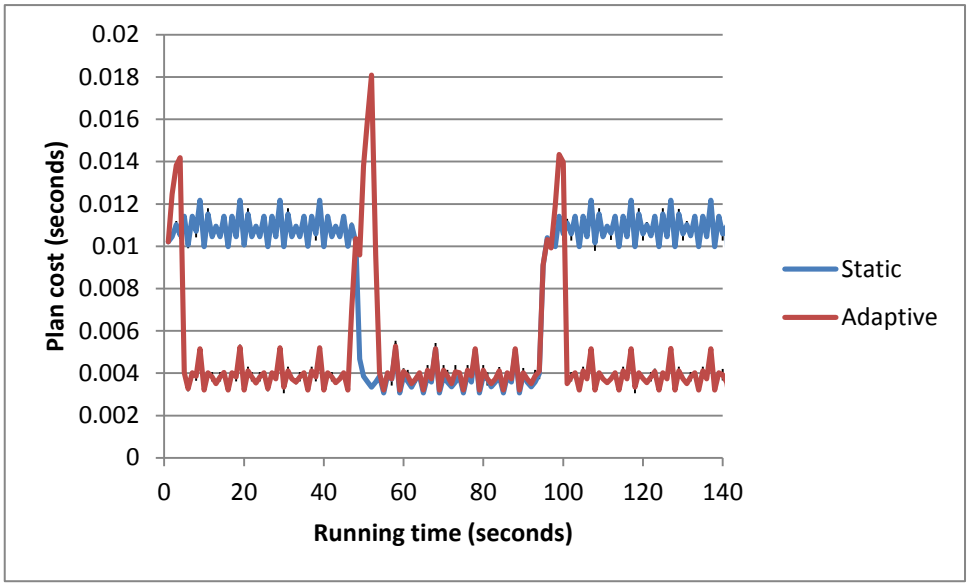


Figure 7.19: Adaptivity while changing stream selectivities

7.4 Conclusion

This chapter has presented experiments that investigate the trade-off between window sizes and completeness of results; advantages of operator sharing, along with a thorough evaluation of R4's adaptive optimiser.

Observing the increased completeness of results and increased processing time in correlation with increasing windows sizes in the first experiment (Section 7.1) positively supports our second hypothesis.

> *Hypothesis 2:* It was expected that the trade-off between the completeness of the output results and the processing time could be controlled by varying the resource allocation.

Second, the increase of total window sizes in the unshared plan over the shared one in the second experiment (Section 7.2) positively supports our third hypothesis.

> *Hypothesis 3:* It was anticipated that resource usage would be reduced by our approach of enabling node sharing, where possible, between the rules.

Finally, the different experiments carried out in sections 7.3.2 and 7.3.3 show the effectiveness of the adaptive optimiser in maintaining lower plan costs – even under changing conditions – which positively supports our fourth hypothesis.

Hypothesis 4: It was anticipated that system performance would be improved by monitoring the characteristics of the streams in order to re-organise the reasoning networks.

However, there are two main factors that we have not considered in our evaluation of the adaptive optimiser. First, our evaluation is based on only two rules, which may have introduced a degree of bias into the results; future evaluation could improve on this by considering a wider range of rules that correspond to other use cases that go beyond the use case considered. Secondly, while we considered the overhead associated with plan migration, the overhead associated with monitoring has not been evaluated. Although it can be controlled by minimising and maximising the monitoring interval (which affects the adaptivity), the monitoring overhead should be evaluated and added to the cost of adaptive optimisation.

Chapter 8: Conclusions and Future Work

The work presented in this thesis falls within the semantic stream reasoning domain, which aims to integrate semantic reasoning techniques with data streams. This can potentially fill the gap between the IoT paradigm (where data streams are generated and processed, but with no standard format or semantic reasoning abilities) and the Semantic Web area (where reasoners work on standardised semantic data efficiently, but not at a high change rate, as in streams).

The primary aim of this research was to enable efficient rule-based reasoning over RDF data streams using dataflow networks, where reasoning is implemented natively over streams using data flow networks. We also addressed the optimisation issue by enabling the reasoning networks to adaptively change at run time, in order to cope with their environmental conditions. A summary of the research and contributions is provided in Section 8.1, and possible improvements recommended for future work in Section 8.2.

8.1 Summary

Several requirements were identified that were relevant to our main research question (outlined in Chapter 1):

> How could we efficiently (by using minimal resources and ensuring high throughput) and effectively (by providing timely results with high precision and recall) enable rule-based reasoning over RDF data streams using data flow networks?

The requirements included integrating the streaming and static data, ensuring low latency, utilising memory, supporting inference, and managing a dynamic environment. The primary objective of the research for this thesis was to create R4, a rule-based reasoner for RDF streams, using the Rete algorithm. R4 is based on a continuous reasoning framework that addresses the stated requirements as follows. R4 tackles the integration requirements by enabling unified and native processing of the static and streaming RDF data. While this feature made it necessary to implement R4 from scratch, rather than reuse existing semantic reasoners and stream processing systems (as in the early semantic stream processing systems, such as C-SPARQL), there are two main advantages to this approach. Firstly, it avoids the overheads involved in transforming the RDF streams to the underlying

stream engine model. Secondly, it involves full control over the low-level processing plans, enabling better optimisation opportunities.

R4 supports background reasoning and domain, user-defined rules using Rete networks in a unified way. This helps to address the inference support requirement. The incremental reasoning enabled by the Rete algorithm helps to keep the latency to a minimum, avoiding costly re-computation. As intermediate results need to be maintained using this approach, we addressed the memory utilisation problem by using the windowing technique. Every partial or complete result was assigned an expiration time so that resource usage was kept under control as the engine continuously removed expired elements. The resource utilisation requirement was further addressed as the intermediate results between the different plans could be shared.

R4 supports adaptive optimisation to further improve performance, and to address the requirement of managing a dynamic environment. As stream characteristics change at run time, what was at first an optimal plan may perform increasingly poorly. The running plan in R4 can be adapted to a more efficient one using a cost-based model that is specifically designed for RDF streams.

After testing the implemented R4 system using a number of use cases, we conducted a comparative evaluation of its performance, first comparing it to a static reasoner, proving our first hypothesis that a stream reasoner would outperform a static reasoner in terms of throughput and response time. We then compared R4 to two stream reasoning engines, Sparkwave and Etalis, and discovered that R4 outperformed them both. Lastly, we evaluated the performance of the adaptive optimiser using different settings (stable and changing). The adaptive optimiser generated more cost-effective plans than those generated by a static optimiser for all of these settings.

8.1.1 Contributions

R4, a continuous rule-based processing reasoner that works natively on RDF streams, was the main contribution to this thesis. We applied the incremental Rete algorithm to overcome the challenge of RDF stream reasoning, and tackled issues such as memory management and sharing memories of different window sizes. R4 creates Rete networks for background reasoning entailment rules and domain-specific, user-defined rules in the same way and connects the first to the latter. The results from both networks were assigned expiration times and re-entered into the networks to generate further results. This is

different to the way in which Sparkwave (Komazec et al., 2012) applies the Rete algorithm to the problem. As Sparkwave does not support reentrancy, it handles background reasoning using a schema pre-processing step and an additional network (ε-network) that produces entailments to be used by the main Rete network. INSTANS (Rinne et al., 2012a) also uses the Rete networks to incrementally process RDF streams. However, it does not discuss the challenge of background reasoning. In addition, the insertion and removal of RDF statements is handled using explicit INSERT and DELETE queries, unlike our garbage collection approach embedded in the beta nodes.

Our second contribution is the cost-based adaptive optimiser. We based our cost model on the body of work produced by the stream management community, while undertaking RDF-specific issues, such as selectivity estimation. Except for CQELS (Le-Phuoc et al., 2011), the semantic stream processing engines reviewed in Chapter 3 do not support adaptive optimisation. On the other hand, CQELS employs an Eddies-based adaptivity approach at the very fine-grained level of triples. We believe that our more coarse-grained adaptivity approach at the level of plans is more suited to the RDF model as it avoids the overhead of adding routing information to every triple.

The extension of Rule Interchange Format (RIF) Core was another minor contribution to our research. Adding window constructs enables users to define time constraints to be used by the processing engine. These windows can be defined at import level, which works as a global window across all the rules in the document, or at formula level, creating local windows that override global ones.

8.2 Future Work

This research focused on enabling efficient rule-based reasoning for RDF streams. We support reasoning for rules that can be expressed in the RIF Core rule language. To enhance efficiency, we applied the incremental Rete algorithm and supported adaptive optimisation to keep the performance at a high level, even in dynamic conditions. This work could be further improved in three main directions, namely supporting the adaptive control of the window size, increasing expressivity by assisting non-monotonic reasoning, and increasing efficiency and scalability by supporting distributed processing.

8.2.1 Adaptive window size

R4's adaptive optimiser is capable of changing the processing plan at run time to a more efficient plan based on a cost model. However, in some cases (e.g. stream bursts), even the most cost-effective plan does not result in adequate performance. In devising a plan cost using a cost model in terms of the time needed to process a number of elements equal to the input rate, with the input rate being defined as the number of incoming triples per second, any plan cost of ≥ 1 second is considered to be inefficient. For example, if a stream input rate is 2000 triples/second, where the cheapest plan cost is 2 seconds, then at the end of the first second, the plan would have processed only half of what had arrived at the beginning, but would be faced with another 2000 triples arriving at second 2. The accumulation of unprocessed elements means that the system loses its responsiveness.

In this case, maintaining responsiveness may be considered more important than completeness of the results. Therefore, a feasible solution is to apply window reduction. In Section 7.1, we saw how the trade-off between the completeness of results and processing times could be controlled by the window sizes. We suggest that the optimiser should be enabled to adaptively control the window sizes in order to handle situations of data overflow.

Reducing the window sizes can be carried out globally, or at local operator level. Global window reduction can be compared to other load shedding techniques, such as sampling, as effectiveness may vary between the approaches. The number of results and memory and processing costs were compared when applying window reduction and sampling in Cammert et al. (2008). They found that the window reduction technique returned more results than the sampling technique.

Furthermore, window reduction can be performed locally in some nodes. In this case, nodes that contribute little to the final output results should have reduced window sizes. To apply this technique, the monitor should be extended to continuously measure how each node's output extends throughout the network. This can be especially beneficial with ontological background reasoning as R4 follows a forward chaining approach, whereby many entailed results are not actually used.

8.2.2 Expressivity

The expressivity of R4 could be improved by supporting the more expressive dialect of RIF, namely RIF Production Rule Dialect (PRD) (de Sainte Marie et al., 2013). RIF PRD

extends RIF Core mainly by allowing different forms of actions in the head part of the rule, including Assert, Retract, and Modify, thus enabling non-monotonic reasoning. RIF PRD formulas also allow negation, which is not supported by RIF Core. In its current form, our continuous rule-based reasoning framework cannot support negation and non-monotonic reasoning as retraction is only supported for expired results. As reasoning in our framework is incremental, retracting a triple or a token before it expires means that all subsequent nodes and memories should be notified to retract any elements that were produced as a result of the presence of that triple. As we used the time intervals with an expiration time approach, each join node independently removed expired elements from its parent memories without communicating with the other nodes. Therefore, invalidating an element before its expiration time would result in incorrect answers (false positives).

The negative tuples approach is a different way of removing expired elements found in the literature of stream processing which avoids reliance on timestamps (Hammad et al., 2003). According to this approach, a window operator emits a positive tuple for every arriving tuple, and a negative tuple (effectively the same tuple but with a negative sign) when this tuple expires. All subsequent operators should be able to deal with positive and negative tuples. Negative tuples can be used not only to ask for the removal of expired tuples, but also to retract invalid tuples. However, the main drawback to this approach is that it doubles the number of tuples going through the network, which reduces the efficiency of the system. We expect this to affect RDF stream processing to a higher extent (than relational streams) because of the fine-grained nature of the RDF model. For instance, a simple relational to RDF mapping tool would generate five triples for a single relational tuple with five attributes. Therefore, instead of undergoing a complete shift to this approach in order to support retraction, we propose the adoption of a hybrid method in which the current expiration timestamp approach is retained and negative tuples only used when it is necessary to invalidate an element before it expires. All operator algorithms should be updated with instructions for operators to follow upon receiving a negative tuple.

8.2.3 Distribution

An important challenge in stream processing in general is distributed processing. Data streams are usually distributed in nature. For some high volume streams produced by widely distributed sources, simply collecting all the streams to be processed in a single machine can be inefficient (Babcock et al., 2002). Performing some processing on data locally at the source (e.g. filtering) can improve performance by reducing expensive data

transfer costs (e.g. SNEE (Galpin et al., 2011)). The ability to distribute processing over multiple machines also enables more scalable systems, as they can scale in two dimensions: the hardware performance of each computing node and the number of nodes (Urbani et al., 2009)). A distributed stream reasoner should be also more fault-tolerant than a centralized reasoner, since it avoids a single point of failure and enables the migration of operators between the affected nodes. Furthermore, overloading problems can be avoided by supporting automatic load balancing over the available nodes.

We suggest a distributed system architecture for R4, in which data flow networks can be distributed among multiple machines that can communicate and dynamically share load with each other. Figure 8.1 shows a dataflow network distributed among several hosts. To enable this, we discuss the communication and load management problems.

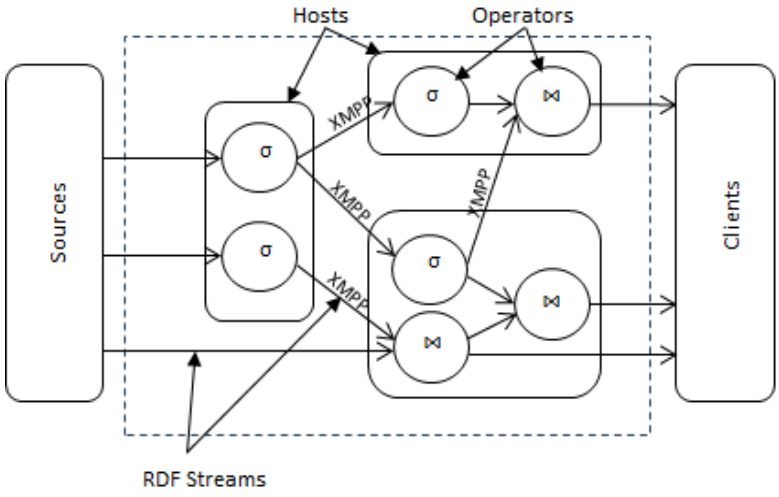


Figure 8.1: A distributed dataflow network

8.2.3.1 Communication Framework

Communication on the Web is based on the request-response approach using the HTTP protocol. In this approach, a client asks for information from a server and the server responds by submitting the information to the client. This protocol is sufficient for downloading static documents. However, modern Web applications need a more dynamic approach, as HTTP does not support real time communication where the server can automatically push updates to clients. To overcome these drawbacks, many protocols have

been developed to support transportation of streaming and real time data. These include RTSP (Schulzrinene et al., 1998), XMPP (Saint-Andre, 2004), Pubsubhubbub[24], MQTT[25]. Several factors should be considered to determine which protocol to use:

- Distribution style: a push-based protocol will better satisfy the real-time requirements
- Latency: the time it takes for data to propagate between the network nodes should be as low as possible
- The underlying transport protocol: HTTP-based protocols will be more suitable for Web applications
- Implemented libraries and active developer communities

The Real Time Streaming Protocol (RTSP) can be excluded for using a pull-based distribution style and its higher latency, as it requires at least three request-response sequences. While the other three protocols support a publish/subscribe communication, MQTT can also be excluded for not being HTTP-based. Pubsubhubbub is a simple, pub/sub, HTTP-based protocol with latency kept at minimum. However, it appears to have fewer implementations and libraries than XMPP. The eXtensible Message and Presence Protocol (XMPP) is an open protocol for message-oriented middleware which can provide near real time communication. It has many features that satisfy the above requirements: push-based distribution style, minimum latency, Web based protocol, and widely deployed and tested libraries for both server and client sides. It is actually a decentralised system; anyone can run their own XMPP server.

Using XMPP as a communication protocol, each host runs as an XMPP server and client at the same time. These hosts subscribe to their preceding nodes in the network, as instructed by the optimiser. The interaction framework involves two stages: the handshaking stage and the data streaming stage. The first stage is more complicated and expensive than the second one, but it only happens once, while the lightweight second stage is long-lasting.

8.2.3.2 Dynamic Load Management

For a distributed dataflow network to perform efficiently, load needs to be balanced between the multiple hosts so that the processing capacity is used to its fullest advantage. Random allocation of tasks may result in some hosts being overloaded while other hosts are idle or lightly loaded. A load distribution mechanism is needed to transfer tasks from

[24] Available at: <http://pubsubhubbub.appspot.com/>

[25] Available at: <http://mqtt.org/>

the poorly performing, overloaded hosts to lightly loaded hosts, so that tasks can take advantage of resources that would otherwise go unused. Furthermore, the dynamic characteristics of streaming data require an adaptive load balancing algorithm that takes account of the runtime changing system-state information.

As in the adaptive optimisation mechanism, a central network optimiser can be used to make decisions of when and where to move loads between the participating hosts. These decisions are to be based on the run-time performance and load statistics of each host collected and periodically reported by local monitors of these hosts. As in the Borealis correlation load balancing algorithm (Abadi et al., 2005), we can define the load in terms of CPU utilisation for each host and operator. For a specific period of time, the load of an operator is the fraction of the CPU time needed by that operator over the length of the period. In other words, if the average tuple arrival rate in period *i* for operator *o* is $\lambda(o)$ and the average tuple processing time for operator *o* is $p(o)$, then the load of *o* in period *i* is $\lambda(o)\,p(o)$. The load of a host in a given period is defined as the sum of all its operators' loads in that period.

We move now from the information policy of the load management mechanism to the transfer policy, i.e. deciding which hosts are suitable to participate in a task transfer, either as senders or as receivers. As used in both Flux (Shah et al., 2003) and the Borealis correlation algorithm, a relative pairwise policy can be followed. After the central optimiser receives information about the loads of all hosts, it orders the hosts by their average load. Then the first (the most loaded) host is paired with the last (the least loaded) host and the second host with the penultimate host, and so on, such that the i^{th} host in the ordered list is paired with the $(n\text{-}i+1)^{th}$ host in the list. For each pair, the optimiser considers moving load from the first host to the second. However, to minimise the load migration overhead—which can nullify the possible benefits of the redistribution—some threshold tests can be applied for each pair before deciding to move load. If the donor's load is less than the average load of all hosts, or if the load difference between the two hosts is less than a predefined threshold, or if the receiver's load is above a threshold, then this pair is not considered for a load redistribution, and the optimiser moves to the next pair. When a pair is selected for load migration, operators are selected from the donor host to be moved based on their individual loads, such that the selected operator's total load is less than *(donor's load – receiver's load)/2*. Then the load migration process starts.

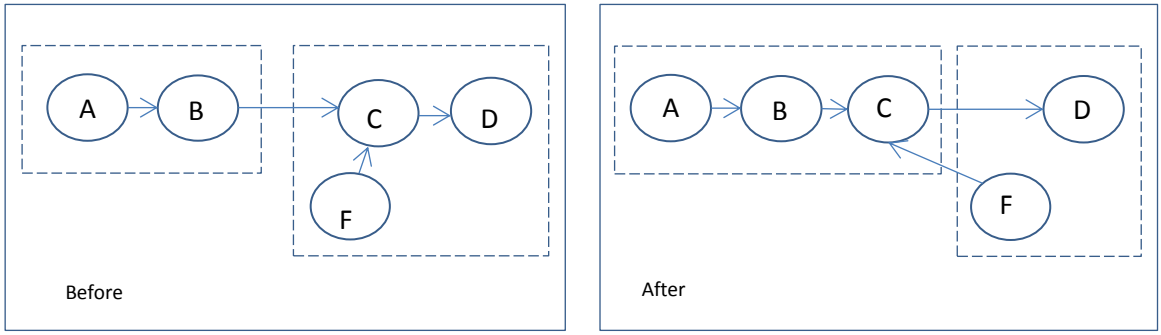


Figure 8.2: Moving operator upstream

The optimiser starts the load migration process by asking the donor host to suspend the execution of the selected operators to be moved. At the same time, the optimiser also asks the receiver host to instantiate these operators and provides it with the addresses of their predecessors so they can subscribe to them. The donor host is then asked to send the states of these operators to the receiver host. Once the transfer is complete, the optimiser sends alerts to the successor operators in the original plan to subscribe to the new locations and then asks the donor host to delete the moved operators.

As the above mechanism does not consider bandwidth issues, a simple optimisation technique for a distributed data flow network can also be employed as follows: For any connection with limited bandwidth, or with traffic bigger than a predefined threshold, we consider moving the sending operator on the edge downstream, or moving the receiving operator on the other end of the connection upstream. First, we check the output throughput of the sending operator and compare it to the throughput of its predecessor. If the sender produces more triples than its predecessor, then the sender should be moved downstream. If it produces fewer triples, then we compare it to the receiver's throughput. If the receiver produces less output than the sender, then we move the receiver upstream.

This technique is similar to the "Box Sliding" technique briefly described in Cherniack et al., (2003), where an operator with a selectivity value of more than one is moved downstream, while operators with low selectivity are moved upstream. However, in its simple mechanism, this technique might only perform well—i.e. reduce network traffic—if both the sending and the receiving operators have only one input. If either one of them has other inputs then moving it to another site without taking consideration of the other inputs may increase network traffic. An example is illustrated in Figure 3.4. Here, the receiving operator C has another input on the same host. Moving C to Host1 because C is more

selective than B will cause F to send its output using the same connection. A carefully designed load balancing algorithm is needed that takes these cases into account.

Appendices

Appendix A RDFS++ Background Reasoning

RDFS++ reasoning supports the main RDFS predicates (domain, range, subPropertyOf, and subClassOf) in addition to a number of lightweight but useful OWL predicates (sameAs, inverseOf, and TransitiveProperty). In this appendix, we present the entailment rules used to reason over these predicates, followed by a shared Rete network to evaluate these rules.

```
(* rdfs2 *)
Forall ?x ?p ?y ?c(
  If  And( ?p [rdfs:domain -> ?c]
           ?x [?p ->?y])
  Then ?x [rdf:type -> ?c])

(* rdfs3 *)
Forall ?x ?p ?y ?c(
  If  And( ?p [rdfs:range -> ?c]
           ?x [?p ->?y])
  Then  ?y [rdf:type -> ?c])

(* rdfs5 *)
Forall ?x ?y ?z(
  If  And( ?x [rdfs:subPropertyOf -> ?y]
           ?y [rdfs:subPropertyOf -> ?z])
  Then  ?x [rdfs:subPropertyOf -> ?z])

(* rdfs6 *)
Forall ?x ?p ?y ?q(
  If  And( ?p [rdfs:subPropertyOf -> ?q]
           ?x [?p -> ?y])
  Then  ?x [?q -> ?y])

(* rdfs9 *)
Forall ?x ?y ?a(
  If  And( ?x [rdfs:subClassOf -> ?y]
           ?a [rdf:type -> ?x])
  Then  ?a [rdf:type -> ?y])

(* rdfs11 *)
Forall ?x ?y ?z(
  If  And( ?x [rdfs:subClassOf -> ?y]
           ?y [rdfs:subClassOf -> ?z])
  Then  ?x [rdfs:subClassOf -> ?z])

(* owlinv *)
Forall ?x ?p ?y ?q(
  If  And(?x [?p -> ?y]
          ?p [owl:inverseOf -> ?q])
  Then  ?y [?q -> ?x])

(* owlinv2 *)
Forall ?p ?q(
  If ?p [owl:inverseOf -> ?q]
  Then  ?q [owl:inverseOf -> ?p])
```

```
(* owltra *)
Forall ?x ?p ?y ?z(
  If  And(?x [?p -> ?y]
          ?y [?p -> ?z]
          ?p [rdf:type -> owl:TransitiveProperty])
  Then  ?x [?p -> ?z])

(* owlsame *)
Forall ?x ?p ?y ?z(
  If  And(?x [?p -> ?y]
          ?x [owl:sameAs -> ?z])
  Then  ?z [?p -> ?y])

(* owlsame2 *)
Forall ?x ?y(
  If ?x [owl:sameAs -> ?y]
  Then  ?y [owl:sameAs -> ?x])
```

Listing A.1: RDFS++ rules in RIF Core

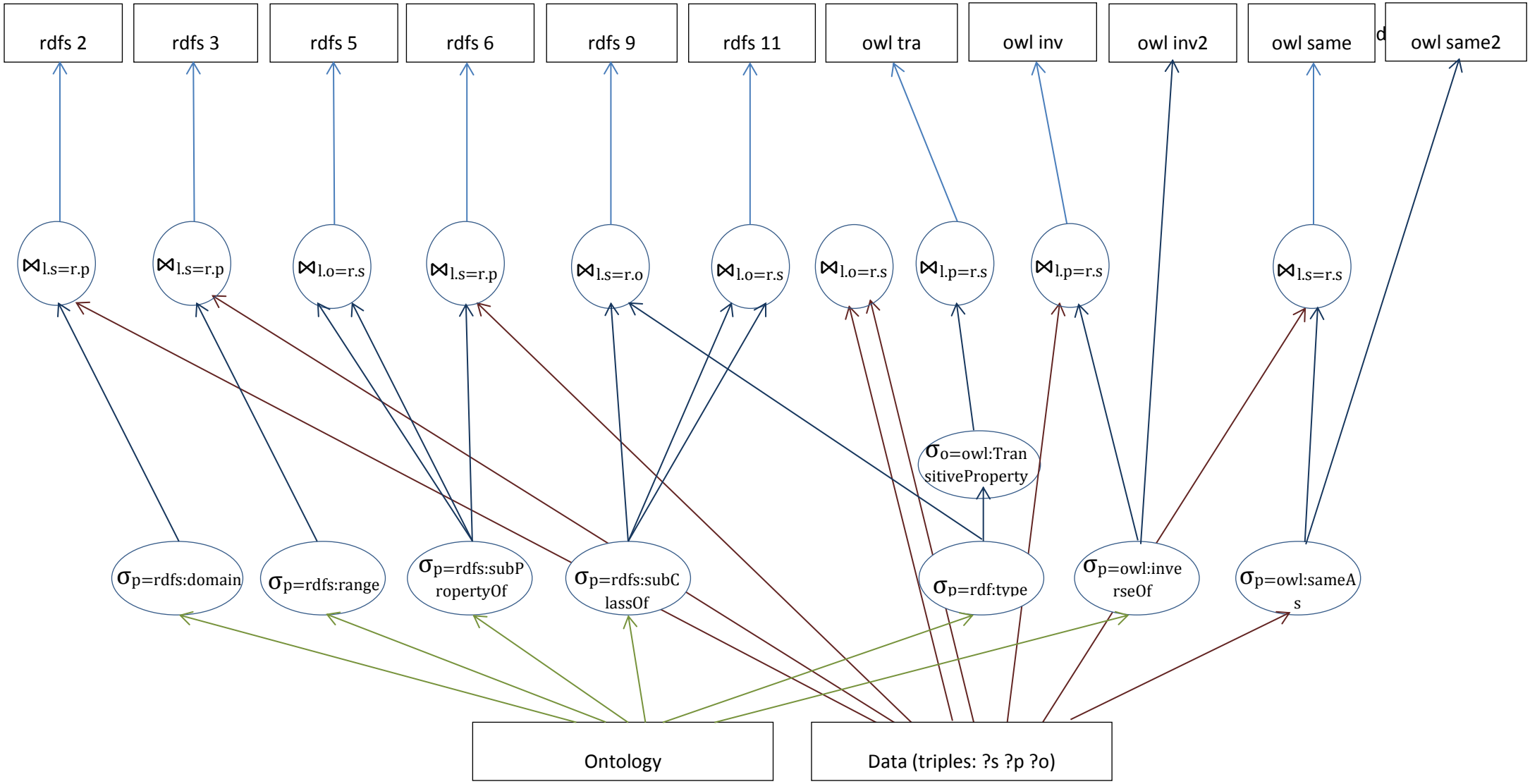


Figure A.1: RETE network for RDFS++ rules

Appendix B Raw Comparative Evaluation Data

This appendix contains the raw evaluation data for some of the experiments described in Chapter 5. Section B.1 contains the processing time results for the first and second experiments described in Section 5.2.1, while section B.2 contains the processing times of systems in the experiments of Section 5.2.2.

B.1 R4 vs. Jena vs. JenaRete

First experiment

System	Dataset size (triples)	Processing time (seconds)					Average	Standard deviation
		Run 1	Run 2	Run 3	Run 4	Run 5		
	2366	0.31	0.28	0.27	0.29	0.27	0.29	0.02
	14098	0.55	0.48	0.47	0.49	0.47	0.49	0.03
	114562	1.76	1.86	1.76	1.61	1.65	1.73	0.10
	462560	5.80	5.34	5.49	6.07	5.60	5.66	0.29
Jena	1121974	14.49	14.49	14.31	14.60	14.25	14.43	0.15
	2366	0.27	0.24	0.23	0.24	0.23	0.24	0.02
	14098	0.46	0.41	0.41	0.41	0.41	0.42	0.03
	114562	1.49	1.47	1.47	1.36	1.41	1.44	0.06
	462560	4.83	4.78	4.85	4.69	4.78	4.79	0.06
JenaRETE	1121974	11.98	13.61	12.74	13.13	12.80	12.85	0.60
	2366	0.18	0.12	0.12	0.12	0.12	0.13	0.03
	14098	0.45	0.36	0.35	0.35	0.36	0.37	0.04
	114562	1.67	1.68	1.65	1.82	1.60	1.68	0.08
	462560	4.05	3.97	3.97	4.12	3.81	3.98	0.12
R4	1121974	8.10	8.03	7.67	8.19	8.17	8.03	0.21

Appendix B

Second experiment

Setting 1:

System	Update number	Response delay (milliseconds)					Average	Standard deviation
		Run 1	Run 2	Run 3	Run 4	Run 5		
	1	188	163	165	165	159	168.00	11.45
	2	70	63	61	64	59	63.40	4.16
	3	49	46	44	44	46	45.80	2.05
	4	45	40	37	34	39	39.00	4.06
	5	36	34	33	32	33	33.60	1.52
	6	34	36	35	34	33	34.40	1.14
	7	42	39	41	39	41	40.40	1.34
	8	52	53	49	50	53	51.40	1.82
	9	51	53	48	49	56	51.40	3.21
	10	33	33	33	29	32	32.00	1.73
	11	32	33	32	29	33	31.80	1.64
	12	33	32	33	28	33	31.80	2.17
	13	33	33	33	30	33	32.40	1.34
	14	34	34	54	53	54	45.80	10.78
	15	49	49	31	29	31	37.80	10.26
	16	44	42	44	45	43	43.60	1.14
	17	32	29	31	25	31	29.60	2.79
	18	31	30	32	28	32	30.60	1.67
	19	36	31	35	30	35	33.40	2.70
	20	29	25	30	25	39	29.60	5.73
	21	279	291	273	341	346	306.00	34.89
	22	214	254	216	231	232	229.40	16.06
	23	214	224	218	223	242	224.20	10.73
	24	210	236	221	206	218	218.20	11.63
	25	178	183	186	185	185	183.40	3.21
	26	182	186	194	193	198	190.60	6.47
	27	174	174	181	166	180	175.00	6.00
	28	228	197	216	229	232	220.40	14.43
	29	244	231	240	246	256	243.40	9.10
	30	162	203	165	175	174	175.80	16.21
	31	162	166	167	168	169	166.40	2.70
	32	165	167	170	168	168	167.60	1.82
Jena	33	176	166	179	182	180	176.60	6.31

	34	162	176	168	171	168	169.00	5.10
	35	163	162	169	164	169	165.40	3.36
	36	162	162	170	161	171	165.20	4.87
	37	187	160	190	178	189	180.80	12.56
	38	209	195	210	202	191	201.40	8.39
	39	174	188	199	166	160	177.40	15.99
	40	174	164	181	162	167	169.60	7.83
	41	172	162	234	159	164	178.20	31.56
	42	188	163	171	166	178	173.20	10.04
	43	159	162	182	178	159	168.00	11.11
	44	164	180	188	165	166	172.60	10.81
	45	165	164	162	165	166	164.40	1.52
	46	167	166	166	168	167	166.80	0.84
	47	171	162	193	181	180	177.40	11.63
	48	160	177	160	185	188	174.00	13.40
	1	205	147	149	151	148	160.00	25.20
	2	48	52	51	57	53	52.20	3.27
	3	38	35	38	45	35	38.20	4.09
	4	32	35	35	37	34	34.60	1.82
	5	38	40	39	40	39	39.20	0.84
	6	27	29	28	27	26	27.40	1.14
	7	26	27	30	29	26	27.60	1.82
	8	38	40	43	38	38	39.40	2.19
	9	30	32	34	32	31	31.80	1.48
	10	31	32	34	32	27	31.20	2.59
	11	34	32	34	33	28	32.20	2.49
	12	30	31	34	35	29	31.80	2.59
	13	30	31	31	33	29	30.80	1.48
	14	54	50	61	59	58	56.40	4.39
	15	24	26	26	27	25	25.60	1.14
	16	37	39	39	37	38	38.00	1.00
	17	23	26	26	24	25	24.80	1.30
	18	25	27	27	26	26	26.20	0.84
	19	30	31	31	27	31	30.00	1.73
	20	24	26	26	21	25	24.40	2.07
	21	41	43	44	45	42	43.00	1.58
	22	38	40	44	40	37	39.80	2.68
JenaRETE	23	62	62	71	60	68	64.60	4.67

	24	37	38	46	36	37	38.80	4.087
	25	42	41	38	37	37	39.00	2.345
	26	38	37	65	34	39	42.60	12.661
	27	37	39	50	65	38	45.80	11.946
	28	36	38	36	54	40	40.80	7.563
	29	37	36	35	37	39	36.80	1.483
	30	35	36	35	35	38	35.80	1.304
	31	37	35	35	34	38	35.80	1.643
	32	36	57	37	39	43	42.40	8.591
	33	41	43	40	41	43	41.60	1.342
	34	41	44	42	42	42	42.20	1.095
	35	41	43	39	50	42	43.00	4.183
	36	40	42	40	39	43	40.80	1.643
	37	41	45	41	42	40	41.80	1.924
	38	39	88	89	39	90	69.00	27.395
	39	82	39	32	32	36	44.20	21.335
	40	32	38	33	29	32	32.80	3.271
	41	32	35	33	30	33	32.60	1.817
	42	34	34	32	29	33	32.40	2.074
	43	32	32	29	30	36	31.80	2.683
	44	34	30	30	28	42	32.80	5.586
	45	31	29	28	28	32	29.60	1.817
	46	31	30	29	27	37	30.80	3.768
	47	30	28	30	29	31	29.60	1.140
	48	28	28	30	28	30	28.80	1.095
	1	113	117	110	114	112	113.20	2.588
	2	44	47	45	30	45	42.20	6.907
	3	21	24	23	22	22	22.40	1.140
	4	18	18	18	18	17	17.80	0.447
	5	20	21	23	21	24	21.80	1.643
	6	14	15	15	14	14	14.40	0.548
	7	14	14	13	12	14	13.40	0.894
	8	20	19	20	19	19	19.40	0.548
	9	12	12	12	13	12	12.20	0.447
	10	13	13	14	13	13	13.20	0.447
	11	20	21	19	19	20	19.80	0.837
	12	12	11	12	12	11	11.60	0.548
R4	13	12	12	12	11	12	11.80	0.447

	14	12	11	11	11	11	11.20	0.45
	15	19	19	20	19	19	19.20	0.45
	16	11	12	12	11	12	11.60	0.55
	17	12	12	12	12	13	12.20	0.45
	18	20	20	21	20	25	21.20	2.17
	19	14	14	13	14	13	13.60	0.55
	20	12	12	11	11	11	11.40	0.55
	21	34	35	38	42	36	37.00	3.16
	22	17	17	17	18	17	17.20	0.45
	23	17	17	17	18	17	17.20	0.48
	24	28	32	27	15	29	26.20	6.54
	25	16	15	15	26	16	17.60	4.72
	26	15	14	15	14	15	14.60	0.55
	27	26	26	14	14	27	21.40	6.77
	28	14	16	28	25	15	19.60	6.43
	29	14	15	16	15	15	15.00	0.71
	30	26	27	13	29	27	24.40	6.47
	31	15	14	26	15	14	16.80	5.17
	32	14	15	14	15	15	14.60	0.55
	33	24	24	25	22	24	23.80	1.10
	34	15	15	14	14	14	14.40	0.55
	35	37	36	37	38	47	39.00	4.53
	36	21	19	20	20	13	18.60	3.21
	37	13	14	13	13	14	13.40	0.55
	38	13	14	13	13	21	14.80	3.49
	39	20	21	20	20	13	18.80	3.27
	40	14	14	15	13	13	13.80	0.84
	41	13	15	14	14	20	15.20	2.78
	42	21	21	21	19	14	19.20	3.03
	43	13	14	14	14	21	15.20	3.27
	44	20	20	13	19	13	17.00	3.67
	45	14	13	21	13	14	15.00	3.39
	46	14	14	14	14	21	15.40	3.13
	47	19	20	13	20	13	17.00	3.67
	48	14	14	20	14	21	16.60	3.58

Appendix B

Setting 7:

System	Update number	Response delay (milliseconds)					Average	Standard deviation
		Run 1	Run 2	Run 3	Run 4	Run 5		
	1	1134	1045	989	1089	1003	1052.00	60.23
	2	922	904	785	952	792	871.00	77.28
	3	1107	1152	1013	1076	1081	1085.80	50.62
	4	550	534	530	523	595	546.40	28.92
	5	1382	1423	1440	1408	1433	1417.20	23.06
	6	1801	1854	1823	1821	1791	1818.00	24.23
	7	1174	1259	1242	1166	1254	1219.00	45.24
	8	1215	1197	1171	1258	1159	1200.00	39.12
	9	1264	1304	1410	1264	1335	1315.40	60.72
Jena	10	1221	1354	1221	1165	1328	1257.80	79.85
	1	871	819	819	895	880	856.80	35.56
	2	591	606	606	738	665	641.20	61.11
	3	758	747	747	837	788	775.40	38.31
	4	387	379	379	436	443	404.80	31.94
	5	671	665	665	677	686	672.80	8.90
	6	657	643	643	1268	672	776.60	274.96
	7	494	485	485	479	477	484.00	6.63
	8	1248	1150	1150	542	1150	1048.00	286.03
	9	589	556	556	557	585	568.60	16.86
JenaRETE	10	491	497	497	517	486	497.60	11.78
	1	658	687	654	672	678	669.80	13.76
	2	464	464	467	481	575	490.20	47.92
	3	319	291	296	284	306	299.20	13.66
	4	524	531	324	568	547	498.80	99.17
	5	387	391	672	407	398	451.00	123.78
	6	331	329	333	349	345	337.40	8.99
	7	671	390	410	418	401	458.00	119.53
	8	372	687	320	719	727	565.00	201.32
	9	313	319	672	325	329	391.60	156.87
R4	10	393	350	312	330	337	344.40	30.44

B.2 R4 vs. Sparkwave and Etalis

Experiment 1: Varying window size

System	Window size	Processing time (seconds)					Average	Standard deviation
		Run 1	Run 2	Run 3	Run 4	Run 5		
Etalis (seq)	0.1	465.72	459.49	455.15	466.94	462.27	461.91	4.78
	1	449.76	440.35	443.13	445.40	454.19	446.56	5.48
	2	446.02	453.88	449.32	448.29	442.39	447.98	4.24
	5	442.49	452.26	447.06	454.82	440.78	447.48	6.06
	10	450.85	452.71	445.60	447.68	451.87	449.74	3.00
Etalis (and)	1	11097.22	11402.08	11243.72	11534.86	11273.36	11310.25	165.87
	2	11353.66	11479.98	11581.19	11959.85	11463.37	11567.61	233.64
	5	11645.02	11711.14	11452.99	11550.24	11611.83	11594.24	98.03
	10	11601.35	11482.45	11526.07	11397.74	11474.88	11496.50	74.64
Sparkwave	0.1	25.64	27.65	26.87	27.40	27.71	27.05	0.86
	1	73.01	78.36	75.08	73.27	72.70	74.48	2.36
	2	103.43	105.41	105.24	103.77	104.01	104.37	0.90
	5	163.84	162.40	161.42	162.91	168.53	163.82	2.77
	10	223.97	210.54	233.86	235.70	223.97	228.81	11.78
R4	0.1	8.86	9.09	8.68	8.81	8.70	8.83	0.17
	1	12.49	12.15	12.32	12.20	12.08	12.25	0.16
	2	13.61	14.70	14.59	14.74	14.00	14.33	0.50
	5	14.66	15.67	15.27	15.55	15.08	15.25	0.40
	10	17.79	18.11	16.93	17.23	16.45	17.30	0.66

Appendix B

Experiment 2: Varying schema size

System	No. of subclasses	Processing time (seconds)					Average	Standard deviation
		Run 1	Run 2	Run 3	Run 4	Run 5		
Etalis (seq)	40	516.11	501.57	521.73	494.61	496.77	506.16	12.08
	200	529.27	529.70	538.59	547.13	531.89	535.31	7.58
	585	523.97	535.20	523.50	519.50	530.18	526.47	6.20
	1100	530.78	541.13	535.80	537.04	512.62	531.47	11.17
Sparkwave	40	149.93	151.81	145.28	146.40	152.43	149.17	3.20
	200	209.04	216.86	220.21	215.87	206.42	213.68	5.74
	585	196.11	199.40	197.07	179.93	196.85	193.87	7.89
	1100	215.04	193.56	199.78	191.44	199.96	199.95	9.23
R4	40	14.22	14.37	13.45	13.45	14.05	13.91	0.43
	200	15.33	16.20	16.74	15.99	16.30	16.11	0.51
	585	14.69	15.68	15.34	14.53	14.97	15.04	0.47
	1100	14.54	14.99	14.64	15.25	14.27	14.74	0.38

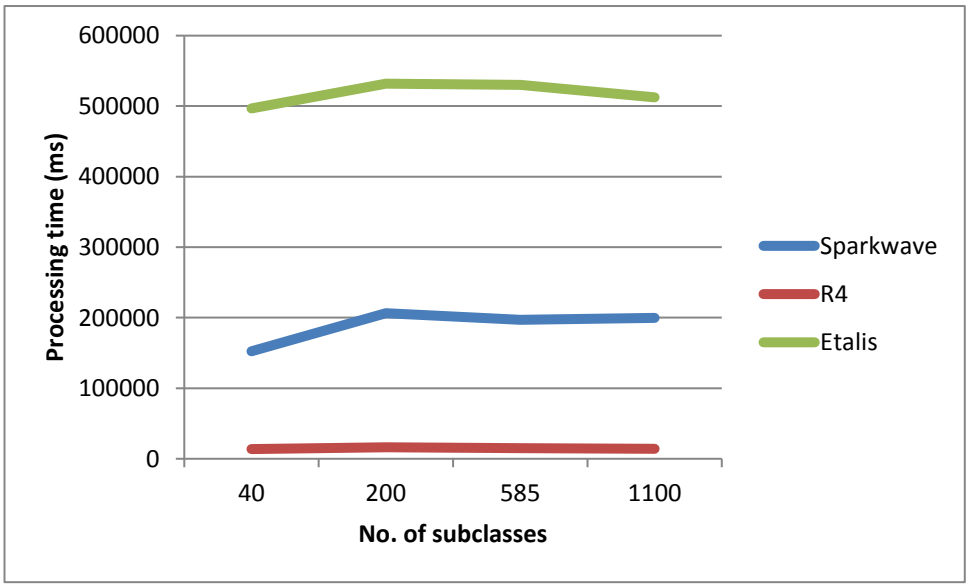


Figure B.1: Processing time of R4, Sparkwave, and Etalis with different schemas

Appendix C Raw Optimisation Evaluation Data

This appendix contains the raw evaluation data for some of the experiments described in Chapter 7. Section C.1 presents the average response time with different window sizes (described in Section 7.1). Section C.2 presents total and individual memory growth (experiments of Section 7.2). Section C.3 has some of the adaptive optimiser experiments (described in Section 7.3).

C.1 Window size vs. completeness

Dataset	Window size	Average response time (milliseconds)					Average	Standard deviation
		Run 1	Run 2	Run 3	Run 4	Run 5		
Ds2	1	2.60	2.40	2.60	2.20	2.80	2.52	0.23
	2	3.00	3.40	2.40	3.00	2.60	2.88	0.39
	4	3.40	3.80	2.80	2.40	2.80	3.04	0.56
	6	4.00	3.60	2.40	2.20	3.80	3.20	0.84
	12	3.40	3.80	2.60	3.80	2.40	3.20	0.66
Ds3	1	4.35	2.61	2.52	2.54	2.59	2.92	0.80
	2	5.26	3.52	3.59	3.61	3.44	3.88	0.77
	4	6.09	4.17	4.37	4.30	4.44	4.67	0.80
	6	7.67	5.54	5.57	5.70	5.09	5.91	1.01
	12	8.54	5.61	5.33	6.50	5.11	6.22	1.40
	24	9.39	6.83	8.67	7.07	8.02	8.00	1.08
	36	9.13	10.04	6.96	7.52	8.67	8.47	1.24
	48	7.59	8.15	5.44	6.11	5.98	6.65	1.16

C.2 Operator sharing

Experiment 1: Total memory growth

Running time	Memories size (triples)	
1	944	1523
2	1891	3049
3	2837	4574
4	3784	6100
5	4730	7625
6	5676	9150
7	6640	10702
8	7569	12201
9	7565	12194
10	7567	12196
11	7583	12221
12	7572	12201
13	7571	12203
14	7581	12216
15	7563	12189
16	7571	12200
17	7563	12189
18	7584	12222
19	7560	12183
20	7574	12209
21	7587	12225
22	7568	12197

Experiment 2: Individual memory growth

time	a1	a2	a3	a4	a5	b1	b2	b3	b4	a6	b5
1	24	169	169	169	123	24	24	48	1	169	24
2	48	338	338	338	249	48	48	96	2	338	48
3	72	507	507	507	374	72	72	144	3	507	72
4	96	676	676	676	500	96	96	192	4	676	96
5	120	845	845	845	625	120	120	240	5	845	120
6	144	1014	1014	1014	750	144	144	288	6	1014	144
7	171	1183	1183	1183	874	171	171	342	8	1183	171
8	192	1352	1352	1352	1001	192	192	384	8	1352	192
9	191	1352	1352	1352	1003	191	191	382	8	1352	191
10	191	1352	1352	1352	1005	191	191	382	8	1352	191
11	194	1352	1352	1352	1003	194	194	388	8	1352	194
12	191	1352	1352	1352	1009	191	191	382	9	1352	191
13	192	1352	1352	1352	1002	192	192	384	9	1352	192
14	193	1352	1352	1352	1007	193	193	386	8	1352	193
15	190	1352	1352	1352	1008	190	190	380	7	1352	190
16	191	1352	1352	1352	1008	191	191	382	9	1352	191

C.3 Adaptive optimisation

C.3.1 Cost model experiments

The table shows the costs of five plans, for each the measured cost (m), then the estimated cost (e).

	P1-m	P1-e	P2-m	P2-e	P3-m	P3-e	P4-m	P4-e	P5-m	P5-e
1	0.00231	0.00239	0.00193	0.00199	0.00177	0.00185	0.00167	0.00174	0.00176	0.00185
2	0.00119	0.00125	0.00097	0.00105	0.00092	0.00098	0.00085	0.00092	0.00092	0.00102
3	0.00137	0.00132	0.00116	0.00113	0.00108	0.00103	0.00103	0.00099	0.00118	0.00108
4	0.00122	0.00130	0.00101	0.00112	0.00093	0.00102	0.00088	0.00099	0.00090	0.00108
5	0.00133	0.00129	0.00113	0.00110	0.00104	0.00100	0.00100	0.00097	0.00112	0.00104
6	0.00132	0.00128	0.00113	0.00109	0.00103	0.00099	0.00099	0.00096	0.00110	0.00102
7	0.00126	0.00127	0.00107	0.00109	0.00098	0.00098	0.00093	0.00095	0.00100	0.00101
8	0.00123	0.00129	0.00104	0.00109	0.00094	0.00100	0.00091	0.00096	0.00093	0.00102
9	0.00119	0.00130	0.00101	0.00110	0.00091	0.00101	0.00088	0.00097	0.00088	0.00105
10	0.00119	0.00134	0.00102	0.00113	0.00091	0.00105	0.00088	0.00099	0.00087	0.00109
11	0.00126	0.00136	0.00101	0.00113	0.00097	0.00108	0.00087	0.00100	0.00094	0.00111
12	0.00125	0.00135	0.00107	0.00114	0.00097	0.00106	0.00094	0.00101	0.00098	0.00111
13	0.00134	0.00133	0.00113	0.00112	0.00105	0.00104	0.00099	0.00099	0.00113	0.00108
14	0.00132	0.00132	0.00108	0.00111	0.00104	0.00104	0.00095	0.00098	0.00107	0.00106
15	0.00129	0.00134	0.00106	0.00113	0.00100	0.00105	0.00092	0.00099	0.00101	0.00109
16	0.00128	0.00134	0.00105	0.00110	0.00099	0.00105	0.00092	0.00097	0.00099	0.00105
17	0.00128	0.00133	0.00106	0.00110	0.00100	0.00104	0.00092	0.00097	0.00100	0.00105
18	0.00130	0.00133	0.00108	0.00111	0.00101	0.00104	0.00095	0.00097	0.00104	0.00106
19	0.00125	0.00134	0.00103	0.00112	0.00096	0.00105	0.00089	0.00098	0.00094	0.00107
20	0.00125	0.00134	0.00104	0.00112	0.00096	0.00105	0.00090	0.00099	0.00095	0.00108
21	0.00130	0.00133	0.00111	0.00112	0.00102	0.00105	0.00097	0.00098	0.00107	0.00107

22	0.00126	0.00134	0.00107	0.00112	0.00098	0.00105	0.00094	0.00099	0.00100	0.00109
23	0.00124	0.00130	0.00104	0.00109	0.00096	0.00101	0.00090	0.00096	0.00095	0.00103
24	0.00129	0.00129	0.00109	0.00108	0.00101	0.00101	0.00095	0.00095	0.00104	0.00101
25	0.00135	0.00129	0.00116	0.00109	0.00106	0.00100	0.00102	0.00096	0.00116	0.00102
26	0.00138	0.00131	0.00115	0.00110	0.00109	0.00103	0.00101	0.00097	0.00119	0.00105
27	0.00130	0.00134	0.00107	0.00113	0.00102	0.00106	0.00093	0.00099	0.00103	0.00109
28	0.00174	0.00140	0.00148	0.00117	0.00145	0.00111	0.00135	0.00104	0.00184	0.00117
29	0.00145	0.00137	0.00121	0.00116	0.00116	0.00108	0.00108	0.00102	0.00132	0.00114
30	0.00136	0.00133	0.00115	0.00113	0.00107	0.00105	0.00101	0.00099	0.00117	0.00109
31	0.00131	0.00136	0.00111	0.00115	0.00102	0.00107	0.00098	0.00101	0.00107	0.00112
32	0.00132	0.00132	0.00110	0.00111	0.00103	0.00104	0.00096	0.00098	0.00108	0.00107
33	0.00147	0.00134	0.00124	0.00111	0.00119	0.00105	0.00110	0.00098	0.00135	0.00106
34	0.00134	0.00133	0.00111	0.00111	0.00105	0.00105	0.00097	0.00097	0.00112	0.00106
35	0.00126	0.00132	0.00103	0.00110	0.00097	0.00104	0.00090	0.00096	0.00096	0.00104
36	0.00121	0.00131	0.00102	0.00110	0.00092	0.00103	0.00088	0.00096	0.00090	0.00105
37	0.00127	0.00134	0.00106	0.00111	0.00098	0.00106	0.00093	0.00098	0.00099	0.00106
38	0.00135	0.00135	0.00113	0.00110	0.00106	0.00106	0.00099	0.00097	0.00113	0.00104
39	0.00137	0.00134	0.00117	0.00112	0.00109	0.00106	0.00104	0.00099	0.00120	0.00109
40	0.00145	0.00135	0.00125	0.00114	0.00117	0.00107	0.00111	0.00100	0.00135	0.00111
41	0.00151	0.00135	0.00131	0.00115	0.00122	0.00107	0.00117	0.00101	0.00146	0.00113
42	0.00143	0.00133	0.00122	0.00113	0.00114	0.00104	0.00109	0.00100	0.00131	0.00110
43	0.00153	0.00135	0.00133	0.00115	0.00125	0.00106	0.00119	0.00101	0.00150	0.00112
44	0.00138	0.00129	0.00116	0.00110	0.00109	0.00100	0.00101	0.00096	0.00121	0.00104
45	0.00155	0.00131	0.00132	0.00111	0.00126	0.00102	0.00118	0.00097	0.00153	0.00106
46	0.00107	0.00091	0.00091	0.00078	0.00087	0.00072	0.00081	0.00068	0.00103	0.00074

Appendix C

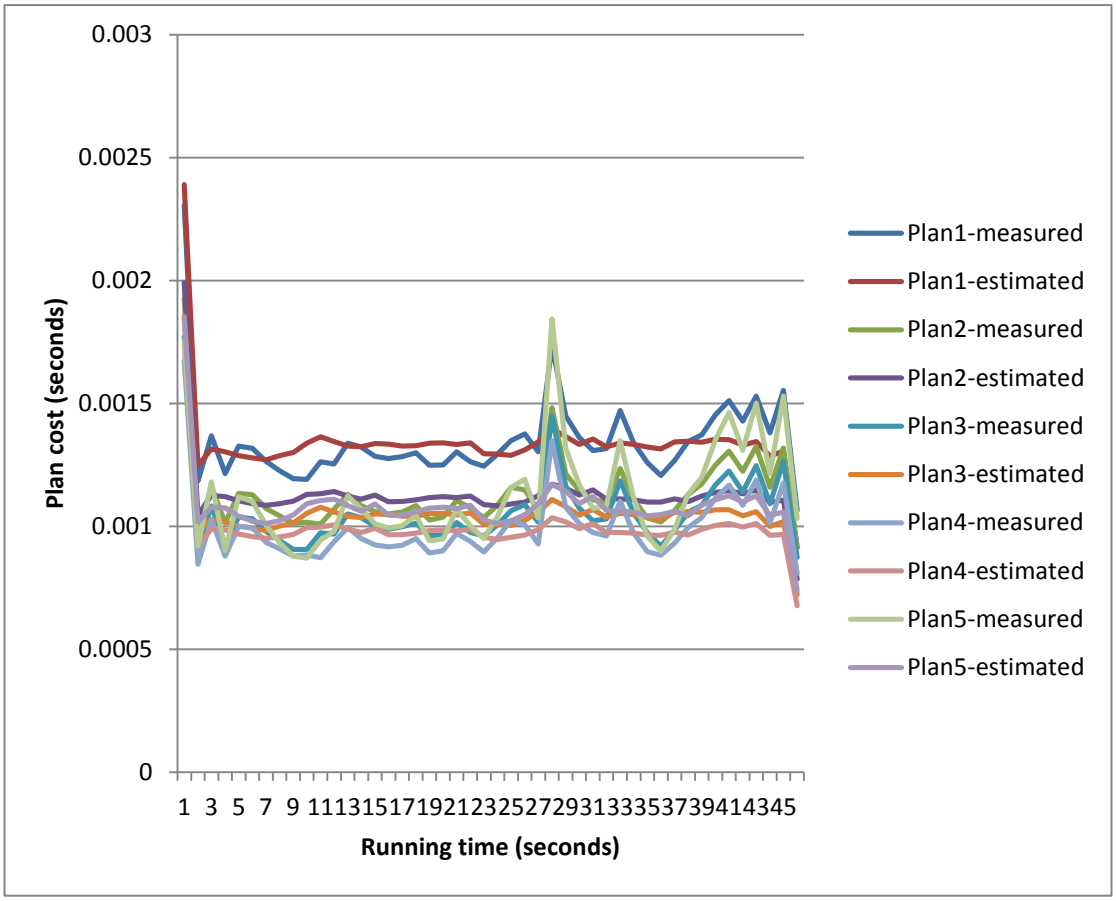


Figure C.1: Measured and estimated costs of five equivalent plans

C.3.2 Comparing static and adaptive plans

Varying window size/Single stream

Plan	Window size (s)	Cost (seconds)					Average	Standard deviation
		Run 1	Run 2	Run 3	Run 4	Run 5		
static	1	3.3E-04	3.3E-04	3.3E-04	3.3E-04	3.3E-04	3.3E-04	1.2E-06
	3	3.6E-04	3.6E-04	3.6E-04	3.6E-04	3.6E-04	3.6E-04	4.3E-07
	5	3.8E-04	3.8E-04	3.8E-04	3.8E-04	3.8E-04	3.8E-04	0.0E+00
	7	3.9E-04	3.9E-04	3.9E-04	3.8E-04	3.9E-04	3.9E-04	1.1E-06
	11	4.1E-04	4.1E-04	4.1E-04	4.1E-04	4.1E-04	4.1E-04	0.0E+00
	15	4.3E-04	4.4E-04	4.3E-04	4.3E-04	4.3E-04	4.3E-04	1.7E-08
adaptive	1	2.1E-04	2.1E-04	2.1E-04	2.1E-04	2.1E-04	2.1E-04	5.2E-07

	3	2.3E-04	2.3E-04	2.3E-04	2.3E-04	2.3E-04	2.3E-04	3.4E-07
	5	2.5E-04	2.5E-04	2.5E-04	2.5E-04	2.5E-04	2.5E-04	0.0E+00
	7	2.5E-04	2.5E-04	2.5E-04	2.5E-04	2.5E-04	2.5E-04	1.3E-07
	11	2.7E-04	2.7E-04	2.7E-04	2.7E-04	2.7E-04	2.7E-04	0.0E+00
	15	2.8E-04	2.8E-04	2.8E-04	2.8E-04	2.8E-04	2.8E-04	1.4E-08
Adaptive with migration	1	2.2E-04	2.1E-04	2.1E-04	2.1E-04	2.1E-04	2.1E-04	7.4E-07
	3	2.4E-04	2.4E-04	2.4E-04	2.4E-04	2.4E-04	2.4E-04	3.0E-07
	5	2.7E-04	2.7E-04	2.7E-04	2.7E-04	2.7E-04	2.7E-04	0.0E+00
	7	2.8E-04	2.8E-04	2.8E-04	2.8E-04	2.8E-04	2.8E-04	2.6E-07
	11	3.1E-04	3.1E-04	3.1E-04	3.1E-04	3.1E-04	3.1E-04	0.0E+00
	15	3.4E-04	3.4E-04	3.4E-04	3.4E-04	3.4E-04	3.4E-04	1.1E-08

Varying input rate

Plan	Rate (t/s)	Cost (seconds)					Average	Standard deviation
		Run 1	Run 2	Run 3	Run 4	Run 5		
	1000	3.8E-04	3.9E-04	3.8E-04	3.9E-04	3.8E-04	3.8E-04	2.5E-06
	3000	1.3E-03	1.3E-03	1.3E-03	1.3E-03	1.3E-03	1.3E-03	0.0E+00
	5000	2.4E-03	2.4E-03	2.4E-03	2.4E-03	2.4E-03	2.4E-03	0.0E+00
	7000	3.3E-03	3.3E-03	3.3E-03	3.3E-03	3.3E-03	3.3E-03	4.8E-19
static	11000	5.9E-03	5.9E-03	5.9E-03	5.9E-03	5.9E-03	5.9E-03	0.0E+00
	1000	2.6E-04	2.6E-04	2.6E-04	2.6E-04	2.6E-04	2.6E-04	5.3E-07
	3000	8.9E-04	8.9E-04	8.9E-04	8.9E-04	8.9E-04	8.9E-04	0.0E+00
	5000	1.6E-03	1.6E-03	1.6E-03	1.6E-03	1.6E-03	1.6E-03	2.4E-19
	7000	2.2E-03	2.2E-03	2.2E-03	2.2E-03	2.2E-03	2.2E-03	0.0E+00
adaptive	11000	3.8E-03	3.8E-03	3.8E-03	3.8E-03	3.8E-03	3.8E-03	4.8E-19
Adaptive	1000	3.1E-04	3.1E-04	3.1E-04	3.1E-04	3.1E-04	3.1E-04	8.7E-07

with migration	3000	1.1E-03	1.1E-03	1.1E-03	1.1E-03	1.1E-03	1.1E-03	0.0E+00
	5000	1.9E-03	1.9E-03	1.9E-03	1.9E-03	1.9E-03	1.9E-03	0.0E+00
	7000	2.7E-03	2.7E-03	2.7E-03	2.7E-03	2.7E-03	2.7E-03	0.0E+00
	11000	4.6E-03	4.6E-03	4.6E-03	4.6E-03	4.6E-03	4.6E-03	0.0E+00

Varying selectivities

Plan	Selectivity factor	Cost (seconds)					Average	Standard deviation
		Run 1	Run 2	Run 3	Run 4	Run 5		
static	0.01	3.8E-04	3.8E-04	3.8E-04	3.8E-04	3.8E-04	3.8E-04	3.3E-06
	0.1	3.8E-04	3.8E-04	3.9E-04	3.8E-04	3.9E-04	3.8E-04	1.7E-06
	0.25	4.0E-04	4.0E-04	4.1E-04	4.0E-04	4.0E-04	4.0E-04	2.2E-06
	0.5	4.3E-04	4.3E-04	4.3E-04	4.3E-04	4.3E-04	4.3E-04	1.9E-06
	0.75	4.7E-04	4.8E-04	4.7E-04	4.7E-04	4.7E-04	4.7E-04	1.5E-06
	1	5.0E-04	5.0E-04	5.0E-04	5.0E-04	5.0E-04	5.0E-04	1.3E-06
adaptive	0.01	2.6E-04	2.6E-04	2.5E-04	2.6E-04	2.6E-04	2.5E-04	1.0E-06
	0.1	2.7E-04	2.7E-04	2.7E-04	2.7E-04	2.7E-04	2.7E-04	1.6E-06
	0.25	3.1E-04	3.1E-04	3.1E-04	3.0E-04	3.0E-04	3.1E-04	1.0E-06
	0.5	3.7E-04	3.6E-04	3.7E-04	3.7E-04	3.7E-04	3.7E-04	1.1E-06
	0.75	4.4E-04	4.4E-04	4.4E-04	4.4E-04	4.4E-04	4.4E-04	1.5E-06
	1	4.9E-04	5.0E-04	5.0E-04	5.0E-04	5.0E-04	5.0E-04	9.3E-07
Adaptive with migration	0.01	3.1E-04	3.1E-04	3.1E-04	3.1E-04	3.1E-04	3.1E-04	2.3E-06
	0.1	3.2E-04	3.3E-04	3.3E-04	3.3E-04	3.3E-04	3.3E-04	1.6E-06
	0.25	3.6E-04	3.6E-04	3.6E-04	3.6E-04	3.6E-04	3.6E-04	1.2E-06
	0.5	4.3E-04	4.3E-04	4.3E-04	4.3E-04	4.3E-04	4.3E-04	8.0E-07
	0.75	5.1E-04	5.1E-04	5.1E-04	5.1E-04	5.1E-04	5.1E-04	8.7E-07
	1	3.1E-04	3.1E-04	3.1E-04	3.1E-04	3.1E-04	3.1E-04	2.3E-06

Varying number of joins

Plan	No. of joins	Cost (seconds)					Average	Standard deviation
		Run 1	Run 2	Run 3	Run 4	Run 5		
static	2	3.6E-04	3.6E-04	3.6E-04	3.6E-04	3.6E-04	3.6E-04	5.3E-07
	3	6.0E-04	6.1E-04	6.1E-04	6.1E-04	6.1E-04	6.1E-04	1.8E-06
	4	8.5E-04	8.6E-04	8.5E-04	8.6E-04	8.6E-04	8.6E-04	1.2E-06
	5	6.4E-04	6.4E-04	6.4E-04	6.4E-04	6.4E-04	6.4E-04	1.1E-06
	6	6.9E-04	6.9E-04	6.8E-04	6.9E-04	6.9E-04	6.9E-04	1.3E-06
adaptive	2	2.3E-04	2.3E-04	2.3E-04	2.3E-04	2.3E-04	2.3E-04	6.8E-07
	3	3.5E-04	3.5E-04	3.5E-04	3.5E-04	3.5E-04	3.5E-04	2.7E-07
	4	4.6E-04	4.6E-04	4.6E-04	4.6E-04	4.6E-04	4.6E-04	1.1E-06
	5	4.7E-04	4.7E-04	4.7E-04	4.7E-04	4.7E-04	4.7E-04	1.3E-06
	6	5.2E-04	5.2E-04	5.2E-04	5.2E-04	5.2E-04	5.2E-04	1.2E-06
Adaptive with migration	2	2.4E-04	2.4E-04	2.4E-04	2.5E-04	2.4E-04	2.4E-04	5.1E-07
	3	3.6E-04	3.7E-04	3.7E-04	3.7E-04	3.7E-04	3.7E-04	8.6E-07
	4	4.9E-04	4.9E-04	4.9E-04	4.9E-04	4.9E-04	4.9E-04	6.2E-07
	5	4.9E-04	4.9E-04	4.9E-04	4.8E-04	4.9E-04	4.9E-04	1.1E-06
	6	5.4E-04	5.5E-04	5.4E-04	5.5E-04	5.5E-04	5.5E-04	1.4E-06

Bibliography

Abadi, D., Carney, D., Çetintemel, 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., Ahmad, Y., Balazinska, M., Çetintemel, U., Cherniack, M., Hwang, J.-H., Lindner, W., Maskey, A., Rasin, A., Ryvkina, E., Tatbul, N., Xing, Y. and Zdonik, S. (2005) The design of the borealis stream processing engine. Paper presented at Second Biennial Conference on Innovative Data Systems Research (CIDR). California, USA 4-7 January.

Abele, A., McCrae, J., Buitelaar, P., Jentzsch, A. and Cyganiak, R. (2017) *Linking Open Data cloud diagram.* Available from: http://lod-cloud.net/ [Accessed 21 April 2017].

Adamer, C., Bannach, D., Klug, T., Lukowicz, P., Sbodio, M., Tresman, M., Zinnen, A. and Ziegert, T. (2008) Developing a wearable assistant for hospital ward rounds: An experience report IN: Floerkemeier, C., Langheinrich, M., Fleisch, E., Mattern, F. and Sarma, S. (eds.) *The Internet of Things*, Springer Berlin Heidelberg, 289-307.

Akyildiz, I.F., Su, W., Sankarasubramaniam, Y. and Cayirci, E. (2002) A survey on sensor networks. *Communication Magazine, IEEE*, 40(8), 102–114.

Ali, M.I., Gao, F. and Mileo, A. (2015) CityBench: A configurable benchmark to evaluate RSP engines using smart city datasets IN: *The Semantic Web-ISWC 2015*. PA, USA, Springer International Publishing, 374–389.

Allemang, D. and Hendler, J (2008) *Semantic Web for the Working Ontologist: Effective Modeling in RDFS and OWL*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Andrade, H.C., Gedik, B. and Turaga, D.S. (2014) *Fundamentals of Stream Processing: Application Design, Systems, and Analytics*. Cambridge University Press.

Anicic, D., Fodor, P., Rudolph, S. and Stojanovic, N. (2011) EP-SPARQL: A unified language for event processing and stream reasoning IN: Proceedings of the 20th International Conference on World Wide Web. March 28 - April 01, 2011 New York, USA, Hyderabad, India: ACM. 635-644.

Anicic, D., Rudolph, S., Fodor, P. and Stojanovic, N. (2012) Stream reasoning and complex event processing in ETALIS. *Semantic Web*, 3(4), 397–407.

Antoniou, G. and Van Harmelen, F. (2004) *A semantic web primer*. MIT press.

Arasu, A., Babcock, B., Babu, S., Datar, M., Ito, K., Motwani, R., Nishizawa, I., Srivastava, U., Thomas, D., Varma, R. and Widom, J. (2003) STREAM: The Stanford stream data manager. *IEEE Data Engineering Bulletin*, 26.

Arasu, A., Cherniack, M., Galvez, E., Maier, D., Maskey, A.S., Ryvkina, E., Stonebraker, M. and Tibbetts, R. (2004) Linear road: A stream data management benchmark IN: *Proceedings of the 30th International Conference on Very Large Data Bases*, Toronto, Canada, 31 August, Vol.30, VLDB Endowment, 480–491,

Arasu, A., Babu, S. and Widom, J. (2006) The CQL continuous query language: semantic foundations and query execution. *The VLDB Journal—The International Journal on Very Large Data Bases, 15*(2), 121–142.

Arpaci-Dusseau, R., Anderson, E., Treuhaft, N., Culler, D., Hellerstein, J., Patterson, D. and Yelick, K. (1999) Cluster I/O with river: Making the fast case common IN: *IOPADS '99: Proceedings of the Sixth Workshop on I/O in Parallel and Distributed Systems*, New York, NY, USA: ACM Press, 10–22.

Atzori, L., Iera, A. and Morabito, G. (2010) The Internet of Things: A survey. *Computer Networks*, 54(15), 2787–2805.

Avnur, R. and Hellerstein, J. (2000) Eddies: Continuously adaptive query processing IN: *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Dallas, Texas, USA, May 16, (Vol. 29, No. 2, pp. 261-272).

Ayad, A.M. and Naughton, J.F. (2004, June) Static optimization of conjunctive queries with sliding windows over infinite streams IN: *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*. Paris, France Jun 13. New York, NY, USA: ACM, 419-430.

Baader, F. (2003) *The description logic handbook: Theory, implementation and applications*. Cambridge university press.

Babcock, B., Babu, S., Datar, M., Motwani, R. and Widom, J. (2002) Models and issues in data stream systems IN: *Proceedings of the 21st ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*. Jun 3 Madison, WI, USA Wisconsin: ACM, 1-16.

Babu, S. and Widom, J. (2001) Continuous queries over data streams. *SIGMOD Record*, 30(3), 109–120.

Babu, S., Motwani, R., Munagala, K., Nishizawa, I. and Widom, J. (2004) Adaptive ordering of pipelined stream filters IN: *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Paris, France. Jun 13. New York, NY, USA: ACM, 407-418.

Babu, S. and Widom, J. (2004) StreaMon: An adaptive engine for stream query processing IN: *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Paris, France. Jun 13. New York, NY, USA: ACM, 931-932.

Babu, S., Munagala, K. and Widom, J. (2005) Adaptive caching for continuous queries IN: *Proceedings of the 21st International Conference on Data Engineering ICDE*. Apr 5. Washington, DC, USA: IEEE, 118–129.

Babu, S. and Bizarro, P. (2005) Adaptive query processing in the looking glass. In *Proceedings of the Second Biennial Conference on Innovative Data Systems Research (CIDR)*, Jan. 2005.

Balduini, M., Celino, I., Dell'Aglio, D., Della Valle, E., Huang, Y., Lee, T., Kim, S. and Tresp, V. (2012) BOTTARI: An augmented reality mobile application to deliver personalized and location-based recommendations by continuous analysis of social media streams. *Web Semantics: Science, Services and Agents on the World Wide Web*, 16(5).

Balduini, M., Della Valle, E., Dell'Aglio, D., Tsytsarau, M., Palpanas, T. and Confalonieri, C. (2013, October) Social listening of city scale events using the streaming linked data framework. In *International Semantic Web Conference* (pp. 1-16). Springer Berlin Heidelberg.

Barbieri, D.F., Braga, D., Ceri, S., Della Valle, E. and Grossniklaus, M. (2009, April) C-SPARQL: SPARQL for continuous querying IN: *Proceedings of the 18th International Conference on World Wide Web*. Madrid, Spain, Apr 20. New York, NY, USA: ACM, 1061–1062.

Barbieri, D., Braga, D., Ceri, S. and Grossniklaus, M. (2010a) An execution environment for C-SPARQL queries IN: *Proceedings of the 13th International Conference on Extending Database Technology*. Lausanne, Switzerland, Mar 22 New York, NY, USA: ACM, 441-452.

Barbieri, D., Braga, D., Ceri, S., Della Valle, E. and Grossniklaus, M. (2010b) Incremental reasoning on streams and rich background knowledge IN: Aroyo, L., Antoniou, G., Hyvonen, E., Ten Teije, A., Stuckenschmidt, H., Cabral, L. and Tudorache, T. (eds.) *The Semantic Web: Research and Applications*. Springer Berlin/Heidelberg, 1-15.

Barbieri, D. and Della Valle, E. (2010c.) A proposal for publishing data streams as linked data IN: *Presented at the International Workshop on Linked Data on the Web (LDOW 2010)*, Raleigh, North Carolina. Apr 27. CEUR Workshop Proceedings, Vol-628.

Barrasa, J., Corcho, O., Gómez-Pérez, A. (2004) R2O, an extensible and semantically based database-to-ontology mapping language. In: *SWDB2004*. 1069–1070

Beck, H., Dao-Tran, M., Eiter, T. and Fink, M. (2015, February) LARS: A logic-based framework for analyzing reasoning over streams. *AAAI*, 1431–1438.

Berners-Lee, T., Fielding, R. and Masinter, L. (1998) RFC2396: Uniform Resource Identifiers (URI). *Generic Syntax*.

Berners-Lee, T., Hendler, J. and Lassila, O. (2001) The Semantic Web. *Scientific American*, 284(5), 28–37.

Berners-Lee, T. (2006) Linked Data - W3C Design Issues. Available from: http://www.w3.org/DesignIssues/LinkedData.html [Accessed 20 December 2016].

Berstel, B. (2002) Extending the RETE algorithm for event management. IN *Temporal Representation and Reasoning, 2002. TIME 2002. Proceedings. Ninth International Symposium*. IEEE. 49-51

Bharathidasan, A. and Sai Ponduru, V. (2003) Sensor networks: An overview. *IEEE Potentials*, 22(2), 20–23.

Biron, P., Malhotra, A. and World Wide Web Consortium, (2004) XML schema part 2: Datatypes. World Wide Web Consortium Recommendation REC-xmlschema-2-20041028.

Bizer, C., Heath, T. and Berners-Lee, T. (2009) Linked data—The story so far. *International Journal on Semantic Web and Information Systems (IJSWIS)*, 5(3), 1–22.

Bizer, C. and Schultz, A. (2009) The berlin SPARQL benchmark. *International Journal on Semantic Web and Information Systems (IJSWIS)*, 5(2), 1–24.

Boley, H., Tabet, S. and Wagner, G. (2001) Design Rationale for RuleML: A Markup Language for Semantic Web Rules. In *SWWS* (Vol. 1, pp. 381-401).

Boley, H. and Kifer, M., (2013) RIF basic logic dialect. W3C recommendation. Available from http://www.w3.org/TR/rif-bld/ [Accessed 8 January 2017].

Boley, H., Hallmark, G., Kifer, M., Paschke, A., Polleres, A. and Reynolds, D. (2013) *RIF Core Dialect*. W3C Recommendation. Available from: http://www.w3.org/TR/2013/REC-rif-core-20130205/ [Accessed 10 July 2015].

Bolles, A., Grawunder, M. and Jacobi, J. (2008) Streaming SPARQL: Extending SPARQL to process data streams. IN: Bechhofer, S., Hauswirth, M., Hoffmann, J. and Koubarakis, M. (eds.) *The semantic web: Research and applications.* Berlin/Heidelberg: Springer, 448-462.

Bonnet, P., Gehrke, J. and Seshadri, P. (2001) Towards Sensor Database Systems IN: Tan, K.-L., Franklin, M. and Lui, J. (eds.) *Mobile Data Management*. Berlin/Heidelberg: Springer, 3-14.

Botan, I., Derakhshan, R., Dindar, N., Haas, L., Miller, R.J. and Tatbul, N. (2010) SECRET: A model for analysis of the execution semantics of stream processing systems. *Proceedings of the VLDB Endowment*, 3(1–2), 232–243.

Brachman, R.J. and Levesque, H.J. (1985) *Readings in knowledge representation*. Morgan Kaufmann Publishers Inc.

Brickley, D., Guha, R. and McBride, B. (2014) *RDF Schema 1.1*. W3C Recommendation. Available from: http://www.w3.org/TR/rdf-schema [Accessed 4 December 2016].

Broekstra, J., Kampman, A. and Van Harmelen, F. (2002) June. Sesame: A generic architecture for storing and querying rdf and rdf schema. In *International semantic web conference*, 54-68. Springer Berlin Heidelberg.

Buchanan, B.G. and Duda, R.O. (1983) Principles of rule-based expert systems. *Advances in computers*, 22, 163–216.

Calbimonte, J.P., Corcho, O. and Gray, A. (2010) Enabling ontology-based access to streaming data sources IN: Patel-Schneider, P.F., Pan, Y., Hitzler, P., Mika, P., Zhang, L., Pan, J.Z., Horrocks, I. and Glimm, B. (eds.) *ISWC 2010, Part I. LNCS*, vol. 6496. Heidelberg: Springer, 96–111.

Cammert, M., Krämer, J., Seeger, B. and Vaupel, S. (2008) A cost-based approach to adaptive resource management in data stream systems. *IEEE Transactions in Knowledge and Data Engineering*, 20(2), 230–245.

Cetintemel, U. (2003) The aurora and medusa projects. *Data Engineering, 51*(3).

Ceri, S., Gottlob, G., and Tanca, L. (1990) *Logical Programming and Databases*. Springer, New York, NY.

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

Chandrasekaran, S., Cooper, O., Deshpande, A., Franklin, M., Hellerstein, J., Hong, W., Krishnamurthy, S., Madden, S., Raman, V., Reiss, F. and Shah, M. (2003) TelegraphCQ: Continuous dataflow processing for an uncertain world IN: *Proceedings of the First Annual Conference on Innovative Database Research (CIDR)*. New York, NY, USA: ACM, 668-668

Chen, C. and Roussopoulos, N. (1994) Adaptive selectivity estimation using query feedback. IN: *Proceedings of the 1994 SIGMOD international conference on Management of data*, May 24-27. ACM, Minneapolis, Minnesota, USA. 161–172.

Cherniack, M., Balakrishnan, H., Balazinska, M., Carney, D., Cetintemel, U., Xing, Y. and Zdonik, S. (2003) Scalable Distributed Stream Processing. *CIDR*, 3, 257–268.

Compton, M., Barnaghi, P., Bermudez, L., GarcíA-Castro, R., Corcho, O., Cox, S., Graybeal, J., Hauswirth, M., Henson, C., Herzog, A. and Huang, V. (2012) The SSN ontology of the W3C semantic sensor network incubator group. *Web Semantics: Science, Services and Agents on the World Wide Web*, *17*, 25–32.

Cugola, G. and Margara, A. (2012) Processing flows of information: From data stream to complex event processing. *ACM Computing Surveys (CSUR)*, *44*(3), 15.

Cyganiak, R. (2005) *A Relational Algebra for Sparql*. HP-Labs Technical Report, HPL-2005-170. Laboratories Bristol.

Cyganiak, R., Wood, D. and Lanthaler, M. (2014) *RDF 1.1 concepts and abstract syntax*. W3C Recommendation. Available from: http://www.w3.org/TR/rdf11-concepts/ [Accessed 06 December 2016].

Da Xu, L., He, W. and Li, S. (2014) Internet of things in industries: A survey. *IEEE Transactions on industrial informatics*, *10*(4), 2233–2243.

Demaine, E., López-Ortiz, A. and Munro, J. (2002) Frequency estimation of internet packet streams with limited space. *Algorithms—ESA 2002*, 11–20.

de Bruijn J., and Welty C. (2013) *RIF, RDF and OWL compatability*. W3C Recommendation. Available from: http://www.w3.org/TR/rif-rdf-owl/ [Accessed 10 March 2016].

de Sainte Marie, C. Paschke, A., and Hallmark, G. (2013) *RIF Production Rule Dialect.* W3C Recommendation. Available from: http://www.w3.org/TR/rif-prd/ [Accessed 03 December 2016].

Della Valle, E., Ceri, S., Barbieri, D.F., Braga, D. and Campi, A. (2008) A first step towards stream reasoning IN: Domingue, J., Fensel, D. and Traverso, P. (eds.) *Future Internet—FIS 2008*. Berlin/Heidelberg: Springer, 72-81.

Della Valle, E., Ceri, S., Harmelen, F.V. and Fensel, D. (2009) It's a streaming world! Reasoning upon rapidly changing information. *IEEE Intelligent Systems, 24*(6), pp.83-89.

Dell'Aglio, D., Calbimonte, J.P., Balduini, M., Corcho, O. and Della Valle, E. (2013) On correctness in RDF stream processor benchmarking IN: *The Semantic Web–ISWC 2013* Berlin Heidelberg: Springer, 326–342.

Dell'Aglio, D., Della Valle, E., Calbimonte, J.P. and Corcho, O. (2014) RSP-QL semantics: A unifying query model to explain heterogeneity of RDF stream processing systems. *International Journal on Semantic Web and Information Systems (IJSWIS)*, 10(4), 17–44.

Dell'Aglio, D. and Della Valle, E. (2014) Incremental reasoning on linked data streams IN: Harth, A., Hose, K. and Schenkel, R. (eds.) *Linked Data Management*. Florida, USA: CRC Press, 413–436.

Deshpande, A., Ives, Z. and Raman, V. (2007) Adaptive query processing. *Foundations and Trends in Databases*, 1(1), 1–140.

Do, T.M., Loke, S.W. and Liu, F. (2011) Answer set programming for stream reasoning IN: *Advances in Artificial Intelligence,* Berlin/Heidelberg: Springer, 104–109.

Domingue, J., Fensel, D. and Hendler, J. (eds). (2011) *Handbook of semantic web technologies*. New York: Springer.

Donini, F.M., Lenzerini, M., Nardi, D. and Schaerf, A. (1996) Reasoning in description logics. *Principles of Knowledge representation*, 1, 191–236.

Dunkels, A. and Vasseur, J. (2008) *IP for Smart Objects*, Internet Protocol for Smart Objects (IPSO) Alliance, White Paper #1.

Fang, S., Da Xu, L., Zhu, Y., Ahati, J., Pei, H., Yan, J. and Liu, Z. (2014) An integrated system for regional environmental monitoring and management based on internet of things. *IEEE Transactions on Industrial Informatics, 10*(2), 1596–1605.

Feldmeier, M. and Paradiso, J. (2010) Personalised HVAC control system IN: *Proceedings of the Internet of Things 2010 Conference*, Tokyo, Japan. Nov 29. IEEE, 1-8.

Finkenzeller, K. (2010) *RFID Handbook: Fundamentals and applications in contactless smart cards, radio frequency identification and near-field communication*. UK: Wiley.

Forgy, C. (1981) OPS5 User's Manual, Report no. CMU-CS-81-135, Department of Computer Science, Carnegie-Mellon University.

Forgy, C. (1982) Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, 19, 17–37.

Frazer, A., De Roure, D., Martinez, K., Nagel, B., Page, K.R. and Sadler, J. (2011) Implementation and Deployment of a Library of the High-level Application Programming Interfaces (SemSorGrid4Env).

Friedman-Hill, E. (2002) *Jess, The Expert System Shell for the Java Platform*. Livermore, California, USA: Sandia National Laboratories, Technical Report SAND98-8206. Version 6.1a3 Available from: http://herzberg.ca.sandia.gov/jess/docs/61/

Galpin, I., Brenninkmeijer, C.Y., Gray, A.J., Jabeen, F., Fernandes, A.A. and Paton, N.W. (2011) SNEE: a query processor for wireless sensor networks. *Distributed and Parallel Databases, 29*(1-2), 31–85.

Garcia-Castro, R., Hill, C. and Corcho, O. (2011) SemserGrid4Env Deliverable D4.3 v2: Sensor network ontology suite. Available from: http://linkeddata4.dia.fi.upm.es/ssg4env/files/deliverables/wp4/D4.3v2.pdf [Accessed 11 May 2017].

García-Castro, R., Corcho, O. and Hill, C. (2012) A core ontological model for semantic sensor web infrastructures. *International Journal on Semantic Web and Information Systems, 8*(1), 22–42.

Garcia-Molina, H., Ullman, J.D. and Widom, J., (2000). Database system implementation (Vol. 654). *Upper Saddle River*, NJ: Prentice Hall.

Gebser, M., Grote, T., Kaminski, R., Obermeier, P., Sabuncu, O. and Schaub, T. (2012) Stream reasoning with answer set programming: Preliminary report. *KR*, 12, 613–617.

Getta, J.R. and Vossough, E. (2004) Optimization of data stream processing. *ACM SIGMOD Record*, 33(3), 34-39.

Ghanem, T.M., Hammad, M.A., Mokbel, M.F., Aref, W.G. and Elmagarmid, A.K. (2007) Incremental evaluation of sliding-window queries over data streams. *IEEE Transactions on Knowledge and Data Engineering*, 19(1), 57–72.

Golab, L. and Ozsu, M. (2003*)* Issues in Data Stream Management. *SIGMOD Record*, 32(2), 5–14.

Gomes, J. and Choi, H.A. (2008) Adaptive optimization of join trees for multi-join queries over sensor streams. *Information Fusion*, 9(3), 412-424.

Gounaris, A., Paton, N., Fernandes, A. and Sakellariou, R. (2002) Adaptive query processing: A Survey IN: Eaglestone, B., North, S. and Poulovassilis, A. (eds.) *BNCOD 2002. LNCS*, vol. 2405. Heidelberg: Springer, 11–25.

Graefe, G. (1990) Encapsulation of parallelism in the volcano query processing system IN: *Proceedings of the ACM SIGMOD international conference on Management of data Conference*, 2, Jun. Atlantic City, N.J. New York, NY, USA ACM,102-111.

Gray, A., Galpin, I., Fernandes, A., Paton, N., Page, K., Sadler, J., Koubarakis, M., Kyzirakos, K., Calbimonte, J.-P., Corcho, O. and Garcia, R. (2009) Semsorgrid4env architecture–phase I. *Deliverable D1. 3v1, SemSorGrid4Env*.

Gray, A., García-Castro, R., Kyzirakos, K., Karpathiotakis, M., Calbimonte, J., Page, K., Sadler, J., Frazer, A., Galpin I. and Fernandes, A. (2011) A semantically enabled service architecture for mashups over streaming and stored data IN: *The semantic web: Research and applications.* Heidelberg: Springer, 300–314.

Grosof, B.N., Horrocks, I., Volz, R. and Decker, S. (2003) Description logic programs: Combining logic programs with description logic IN: *Proceedings of the 12th International World Wide Web Conference (WWW 2003)*. 20-24 May, Budapest, Hungary: ACM 48-57.

Gruber, T. (1993) A Translation Approach to Portable Ontology Specifications. *Knowledge Acquisition*, 199–220.

Guinard, D., Trifa, V., Pham, T. and Liechti, O. (2009) Towards physical mashups in the Web of Things IN: *Proceedings of the 6th International Conference on Networked Sensing Systems*. Pittsburgh, USA. 17-19 June. IEEE, 1-4.

Guinard, D., Trifa, V. and Wilde, E. (2010) A resource oriented architecture for the Web of Things IN: *Proceedings of the Internet of Things 2010 Conference*. Tokyo, Japan. Nov 29. IEEE, 1-8.

Guo, Y., Pan, Z. and Heflin, J. (2005) LUBM: A benchmark for OWL knowledge based systems. *Web Semantics: Science, Services and Agents on the World Wide Web*, 3(2), 158–182.

Gupta, A., Mumick, I.S., Subrahmanian, V.S. (1993) Maintaining views incrementally. *ACM SIGMOD Record, 22*(2), 157–166.

Haarslev, V. and Muller, R. (2001) RACER System Description IN: Gore, R., Leitsch, A. and Nipkow, T. (eds.) *Automated Reasoning*. Berlin/Heidelberg: Springer. Vol. 2083.

Hameed, B., Khan, I. and Durr, F. (2010) An RFID based consistency management framework for production monitoring in a smart real-time factory IN: *Proceedings of the Internet of Things 2010 Conference*, Tokyo, Japan. Nov 29. IEEE, 1-8.

Hammad, M., Aref, W., Franklin, M., Mokbel, M. and Elmagarmid, A. (2003) Efficient execution of sliding-window queries over data streams.

Hammad, M., Mokbel, M., Ali, M., Aref, W., Catlin, A., Elmagarmid, A., Eltabakh, M., Elfeky, M., Ghanem, T., Gwadera, R., Ilyas, I., Marzouk, M. and Xiong, X. (2004) Nile: A query processing engine for data streams IN: *Proceedings of the 20th International Conference on Data Engineering*. Boston, USA, Mar 30. IEEE, 851.

Halpern, J.Y., (2005). *Reasoning about uncertainty*. MIT press.

Hanson, E. and Hasan, M. (1993) *Gator: An optimized discrimination network for active database rule condition testing.* University of Florida, Technical Report TR93-036.

Harth, A. and Decker, S. (2005, November) Optimized index structures for querying RDF from the web. In *Web Congress, 2005. LA-WEB 2005. Third Latin American* (pp. 10-pp). IEEE.

Hartig, O. and Heese, R. (2007) The SPARQL query graph model for query optimization. *The Semantic Web: Research and Applications*, 564–578.

Harris, S., Seaborne, A. and Prud'hommeaux, E. (2013) *SPARQL 1.1 query language*. W3C Recommendation. Available from: http://www.w3.org/TR/sparql11-query/ [Accessed 03 December 2016].

Hebeler, J., Fisher, M., Blace, R. and Perez-Lopez, A. (2009) *Semantic Web Programming*. 1st ed. USA: Wiley Publishing, Inc.

Heinze, T., Jerzak, Z., Hackenbroich, G. and Fetzer, C. (2014, May) Latency-aware elastic scaling for distributed data stream processing systems. In *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems* (pp. 13-22). ACM.

Hellerstein, J., Franklin, M., Chandrasekaran, S., Deshpande, A., Hildrum, K., Madden, S., Raman, V. and Shah, M. (2000) Adaptive query processing: Technology in evolution. *IEEE Data Engineering Bulletin*, 23(2), 7–18.

Hitzler, P., Krötzsch, M., Parsia, B., Patel-Schneider, P.F. and Rudolph, S. (2012) *OWL 2 web ontology language primer*. W3C recommendation. Available from: http://www.w3.org/TR/owl-primer/ [Accessed 05 December 2016].

Hoeksema, J. and Kotoulas, S. (2011) High-performance distributed stream reasoning using S4 IN: *Proceedings of the First International Workshop on Ordering and Reasoning at ISWC 2011*.

Horrocks, I. (2002) DAML+OIL: A Description Logic for the Semantic Web. *IEEE Data Eng. Bull., 25*(1), pp.4-9.

Horrocks, I., Patel-Schneider, P.F., Boley, H., Tabet, S., Grosof, B. and Dean, M. (2004) SWRL: A semantic web rule language combining OWL and RuleML. *W3C Member submission, 21*, p.79.

Ives, Z., Halevy, Y. and Weld, D. (2004) Adapting to source properties in processing data integration queries IN: *Proceedings of ACM SIGMOD International Conference on Management of Data*, Paris, France. Jun 13. New York, NY, USA: ACM, 395-406.

Jang, M. and Sohn, J. (2004) Bossam: An extended rule engine for OWL inferencing IN: Antoniou G. and Boley H. (eds.) *Rules and Rule Markup Languages for the Semantic Web*. Berlin/Heidelberg: Springer. 128-138.

Jara, A., Alcolea, A., Zamora, M., Skarmeta, A. and Alsaedy, M. (2010) Drug interaction checker based on IoT IN: *Proceedings of the Internet of Things 2010 Conference*, Tokyo, Japan. Nov 29. IEEE, 1-8.

Kabra, N. and DeWitt, D. (1998) Efficient mid-query re-optimization of sub-optimal query execution plans. *SIGMOD Conference*. Vol. 27, No. 2, ACM: 106-117.

Kang, J., Naughton, J.F. and Viglas, S.D. (2003, March) Evaluating window joins over unbounded streams IN: *Data Engineering, 2003. Proceedings of the 19th International Conference, Mar 5.* IEE: 341–352.

Kifer, M., Lausen, G. and Wu, J. (1995) Logical foundations of object-oriented and frame-based languages. *Journal of ACM, 42*(4), 741–843.

Kifer, M. and Boley, H. (2013) RIF overview. W3C working group note. Available from http://www.w3.org/TR/rif-overview/ [Accessed 8 January 2017].

Kipf, A. and Kemper, A. (2016) An Integration Platform for Temporal Geospatial Data. *Digital Mobility Platforms and Ecosystems*, p.207.

Komazec, S., Cerri, D. and Fensel, D. (2012, July) Sparkwave: Continuous schema-enhanced pattern matching over RDF data streams IN: *Proceedings of the 6th ACM International Conference on Distributed Event-Based Systems*, Jul 16. NEW YORK, USA: ACM, 58–68.

Kossmann, D. (2000) The state of the art in distributed query processing. *ACM Computing Surveys (CSUR)*, 32(4), 422–469.

Krämer, J. and Seeger, B. (2005) A temporal foundation for continuous queries over data streams IN: *Proceedings of COMAD*, 6-8 January. Goa, India, Computer Society of India 70–82.

Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S. and Scarcello, F. (2006) The DLV system for knowledge representation and reasoning. *ACM Transactions on Computational Logic (TOCL)*, 7(3), 499–562.

Le-Phuoc, D., Dao-Tran, M., Xavier Parreira, J. and Hauswirth, M. (2011) A native and adaptive approach for unified processing of linked streams and linked data IN: Aroyo, L., Welty, C., Alani, H., Taylor, J., Bernstein, A., Kagal, L., Noy, N. and Blomqvist, E. (eds.) *ISWC (2011), Part I. LNCS*, vol. 7031. Heidelberg: Springer, 370–388.

Le-Phuoc, D., Parreira, J. and Hauswirth, M. (2012a) Linked stream data processing IN: *Reasoning web: Semantic technologies for advanced query answering*, Heidelberg: Springer, 245–289.

Le-Phuoc, D., Nguyen-Mau, H.Q., Parreira, J.X. and Hauswirth, M. (2012b) A middleware framework for scalable management of linked streams. *Web Semantics: Science, Services and Agents on the World Wide Web*, 16, 42–51.

Le-Phuoc, D., Dao-Tran, M., Pham, M., Boncz, P., Eiter, T. and Fink, M. (2012c) Linked stream data processing engines: Facts and figures IN: *The Semantic Web–ISWC 2012*. Heidelberg: Springer, 300–312.

Le-Phuoc, D., Quoc, H.N.M., Le Van, C. and Hauswirth, M. (2013) Elastic and scalable processing of linked stream data in the cloud IN: *The Semantic Web–ISWC 2013.* Berlin/ Heidelberg: Springer, 280–297.

Lerner, A. and Shasha, D. (2003) AQuery: query language for ordered data, optimization techniques, and experiments. In *Proceedings of the 29th international conference on Very large data bases-Volume 29* (pp. 345-356). VLDB Endowment.

Liebig, T. and Opitz, M. (2011) Reasoning over dynamic data in expressive knowledge bases with rscale IN: *The 10th International Semantic Web Conference*, October 24. Bonn, Germany.

Lucas, P. and Van Der Gaag, L. (1991) *Principles of expert systems*. Wokingham: Addison-Wesley.

Margara, A., Urbani, J., van Harmelen, F. and Bal, H. (2014) Streaming the web: Reasoning over dynamic data. *Web Semantics: Science, Services and Agents on the World Wide Web*, 25, 24–44.

Matheus, C., Dionne, B., Parent, D., Baclawski, K. and Kokar, M. (2006) BaseVISor: A forward-chaining inference engine optimized for RDF/OWL triples IN: *Proceedings of Digital Proceedings of the 5th International Semantic Web Conference (ISWC 2006)* Nov. 2006. , Athens, GA. 263-271

Manning, C., Raghavan, P. and Schutze, H. (2008) *Introduction to Information Retrieval*. Cambridge University Press.

McBride, B. (2002) Jena: A semantic web toolkit. *IEEE Internet Computing*, 6(6), 55–59.

Mehlhorn, K. and Sanders, P. (2008) *Algorithms and data structures: The basic toolbox*. Springer Science & Business Media.

Metakides, G. and Nerode, A. (1996) *Principles of logic and logic programming*. Elsevier.

Mileo, A., Abdelrahman, A., Policarpio, S. and Hauswirth, M. (2013) Streamrule: A nonmonotonic stream reasoning system for the semantic web IN: *Web Reasoning and Rule Systems*. Berlin/Heidelberg: Springer, 247–252.

Minsky, M. (1975) A Framework for Representing Knowledge, IN: Winston, P.H. (ed.), *The Psychology of Computer Vision*, McGraw-Hill, New York.

Miranker, D. (1987) Treat: A better match algorithm for AI production system matching IN: *Proceedings of AAAI*, California, USA. *AAAI*, 42–47.

Mishra, P. and Eich, M.H. (1992) Join processing in relational databases. *ACM Computing Surveys (CSUR), 24*(1), 63–113.

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

Myung, D., Duncan, B., Malan, D., Welsh, M., Gaynor, M., and Moulton, S. (2002) Vital dust: Wireless sensors and a sensor network for real-time patient monitoring, IN *8th Annual New England Regional Trauma Conference*, Burlington, MA.

Nayak, P., Gupta, A. and Rosenbloom, P. (1988) Comparison of the Rete and Treat Production Matches for Soar IN: *Proceedings of AAAI-88*, 693–699.

Neumann, T. and Moerkotte, G. (2011, April) Characteristic sets: Accurate cardinality estimation for RDF queries with multiple joins. In *2011 IEEE 27th International Conference on Data Engineering. IEEE*. Vancouver. 984-994

Newell, A. (1973). Production systems: models of control structures, in: Chase, W.G. (ed.), *Visual Information Processing*, Academic Press, New York.

Ngai, E., Moon, K., Riggins, J. and Candace, Y. (2008) RFID research: An academic literature review (1995–2005) and future research directions. *International Journal of Production Economics*, 112(2), 510–520. Oliya, M., Zhu, J., Pung, H. and Veillard, A. (2011) Incremental query answering over dynamic contextual information IN: *Tools with Artificial Intelligence (ICTAI), 23rd IEEE International Conference*, Nov 7. IEEE, 452–455.

Paganelli, F. and Giuli, D. (2007, May) An ontology-based context model for home health monitoring and alerting in chronic patient care networks. In *Advanced Information Networking and Applications Workshops, 2007, AINAW'07. 21st International Conference on* (Vol. 2, pp. 838-845). IEEE.

Papataxiarhis, V., Tsetsos, V., Karali, I., Stamatopoulos, P. and Hadjiefthymiades, S. (2009) Developing rule-based applications for the web: Methodologies and tools IN: Giurca, A., Gasevic, D. and Taveter, K. (eds.) *Handbook of research on emerging rule-based languages and technologies: Open solutions and approaches*. Information Science Reference. IGI Global.

Patel-Schneider, P., Hayes, P., Horrocks, I. and van Harmelen. F. (2004*) OWL web ontology language; semantics and abstract syntax*. W3C recommendation. Available from: http://www.w3.org/TR/owl-semantics/ [Accessed 8 July 2014].

Pérez, J., Arenas, M. and Gutierrez, C. (2006, November). Semantics and Complexity of SPARQL. In *International semantic web conference*, 30-43. Springer Berlin Heidelberg.

Polleres, A., Boley, H. and Kifer, M., (2013) RIF datatypes and built-ins 1.0. W3C recommendation. Available from: https://www.w3.org/TR/rif-dtb/ [Accessed 8 January 2017].

Raskin, R.G. and Pan, M.J. (2005) Knowledge representation in the semantic web for Earth and environmental terminology (SWEET). *Computers & geosciences*, *31*(9), 1119–1125.

Ren, Y., Pan, J.Z. and Zhao, Y. (2010) Towards scalable reasoning on ontology streams via syntactic approximation. *Proc. of IWOD*, Aug 23. Shanghai, China.

Ren, Y. and Pan, J.Z. (2011, October). Optimising ontology stream reasoning with truth maintenance systems IN: *Proceedings of the 20th ACM International Conference on Information and Knowledge Management*, Oct 24. ACM, 831–836.

Reynolds, D. (2013) *OWL 2 RL in RIF*. W3C Recommendation. Available from: http://www.w3.org/TR/rif-owl-rl/ [Accessed 10 March 2016].

Rinne, M., Nuutila, E. and Törmä, S. (2012a, November) Instans: High-performance event processing with standard RDF and SPARQL IN: *11th International Semantic Web Conference ISWC 2012 Nov 11*. CEUR-WS. Org, 101.

Rinne, M., Abdullah, H., Törmä, S. and Nuutila, E. (2012b) Processing heterogeneous RDF events with standing SPARQL update rules IN: *On the Move to Meaningful Internet Systems: OTM 2012*. Berlin/Heidelberg: Springer, 797–806.

Saint-Andre, P. (2004) *Extensible messaging and presence protocol (xmpp): Core, RFC*, The Internet Society. Available from: http://tools.ietf.org/html/rfc3920 [Accessed 12 July 2011].

Sakaki, T., Okazaki, M. and Matsuo, Y. (2010, April) Earthquake shakes Twitter users: real-time event detection by social sensors. In *Proceedings of the 19th international conference on World wide web* (pp. 851-860). ACM.

Scales, J. (1986) *Efficient matching algorithms for the SOAR/OPS5 production system*. Computer Science Dept., Knowledge Systems Laboratory: Stanford University, Technical Report KSL 86–47.

Scharrenbach, T., Urbani, J., Margara, A., Valle, E. D., Bernstein, A. (2013) Seven Commandments for Benchmarking Semantic Flow Processing Systems. IN: *Proceedings of the 10th International Conference ESWC2013*. Vol. 7882. 305–319.

Schleier-Smith, J., Krogen, E.T. and Hellerstein, J.M. (2016, December) ReStream: Accelerating Backtesting and Stream Replay with Serial-Equivalent Parallel Processing. In *Proceedings of the Seventh ACM Symposium on Cloud Computing*. ACM, 334–347.

Schmidt, S. (2006) Quality-of-service-aware data stream processing. DEng thesis. TU Dresden. Available from: http://www.qucosa.de/fileadmin/data/qucosa/documents/1895/1174255992231-5474.pdf [Accessed 23 January 2017].

Schmidt, M., Meier, M. and Lausen, G. (2010, March) Foundations of SPARQL query optimization. In *Proceedings of the 13th International Conference on Database Theory*. ACM, 4–33.

Schulzrinne, H., Rao, A. and Lanphier, R. (1998) Real Time Streaming Protocol (RTSP), *RFC 2326*, IETF.

Selinger P., Astrahan M., Chamberlin D., Lorie, R. and Price T. (1979) Access path selection in relational database systems IN: *Proceedings of SIGMOD*. Boston, Mass., USA, 23–34.

Sequeda, J. and Corcho, O. (2009) Linked stream data: A position paper IN: *Proceedings of the 2nd International Workshop on Semantic Sensor Networks (SSN 09)*, Washington, USA.

Shah, M., Hellerstein, J., Chandrasekaran, S. and Franklin M. (2003) Flux: An adaptive partitioning operator for continuous query systems, *ICDE*. In Data Engineering, 2003. Proceedings. 19th International Conference on Mar 5. IEEE, 25-36.

Shah, M.A., Hellerstein, J.M. and Brewer, E. (2004, June) Highly available, fault-tolerant, parallel dataflows. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*. ACM, 827–838.

Sheth, A., Henson, C. and Sahoo, S. (2008) Semantic sensor web. *IEEE Internet Computing* 12(4), 78–83.

Shivaratri N., Krueger, P. and Singhal M. (1992) Load distributing for locally distributed systems. *Computer*, 25(12), 33–44.

Shortliffe, E.H. (1976) *Computer-based Medical Consultations: MYCIN*, Elsevier, New York.

Singh, S. and Karwayun, R. (2010) A comparative study of inference engines IN: *Seventh International Conference on Information Technology: New Generation*, Seventh International Conference, Apr 12. IEEE, 53-57.

Sirin, E., Parsia, B., Grau, B., Kalyanpur, A. and Katz, Y. (2007) Pellet: A practical OWL-DL reasoner. *Journal of Web Semantics*, 5(2), 51–53.

Sowa, J.F. ed. (2014) *Principles of semantic networks: Explorations in the representation of knowledge*. Morgan Kaufmann.

Stocker, M., Seaborne, A., Bernstein, A., Kiefer, C. and Reynolds, D. (2008, April) SPARQL basic graph pattern optimization using selectivity estimation. In *Proceedings of the 17th international conference on World Wide Web*, ACM, 595–604.

Stonebraker, M., Çetintemel, U. and Zdonik, S., (2005). The 8 requirements of real-time stream processing. ACM SIGMOD Record, 34(4), pp.42-47.

Stuckenschmidt, H., Ceri, S., Della Valle, E., van Harmelen, F. and di Milano, P. (2010) Towards expressive stream reasoning IN: *Proceedings of the Dagstuhl Seminar on Semantic Aspects of Sensor Networks.*

Sullivan, M. (1996, September) Tribeca: A stream database manager for network traffic analysis. In *VLDB*, 96, p. 594.

Sullivan, M. and Heybey, A. (1998) Tribeca: A System for Managing Large Databases of Network Traffic IN: *Proceedings of the USENIX Annual Technical Conference*.

Sundamaeker, H., Guillemin, P., Friess, P. and Woelffle, S. (2010) Vision and challenges for realising the Internet of Things. Cluster of European Research Projects on the Internet of Things, European Commission. Brussels.

Taylor, K., Gledhill, R., Essex, J., Frey, J., Harris, S. and De Roure, D. (2006) Bringing chemical data onto the semantic web. *Journal of Chemical Information and Modeling*, 46(3), 939–952.

Terry, D., Goldberg, D., Nichols, D. and Oki, B. (1992) Continuous queries over append-only databases (Vol. 21, No. 2, pp. 321-330). ACM.

Tielert, T., Killat, M., Hartenstein, H., Luz, R., Hausberger, S. and Benz, T. (2010) The impact of traffic-light-to-vehicle communication on fuel consumption and emissions IN: *Proceedings of the Internet of Things 2010 Conference*. Tokyo, Japan. Nov 29. IEEE, 1-8.

Tsarkov, D. and Horrocks, I. (2006) FaCT++ Description logic reasoner: System description IN: Furbach, U. and Shankar, N. (eds.) *Automated Reasoning*. Berlin/Heidelberg, Springer.

Tsialiamanis, P., Sidirourgos, L., Fundulaki, I., Christophides, V. and Boncz, P. (2012) Heuristics-based query optimisation for SPARQL IN: *Proceedings of the 15th International Conference on Extending Database Technology*. ACM, 324–335.

Urbani, J., Kotoulas, S., Oren, E. and van Harmelen, F. (2009) Scalable distributed reasoning using mapReduce IN: Bernstein, A., Karger, D.R., Heath, T., Feigenbaum, L., Maynard, D., Motta, E. and Thirunarayan, K. (eds.) *ISWC 2009. LNCS*, vol. 5823. Heidelberg: Springer, 634–649.

Urhan, T. and Franklin, M. (2000) Xjoin: A reactively scheduled pipelined join operator. *IEEE Data Eng. Bull.* 23(2), 27–33.

Van Rijsbergen, C. (1979) Information Retrieval. Butterworths, London, 1979. Available at: http://www.dcs.gla.ac.uk/Keith/Chapter.7/Ch.7.html [Accessed 24 July 2013].

Vazquez, J. and Lopez-de-Ipina, D. (2008) Social devices: Autonomous artefacts that communicate on the Internet IN: *Proceedings of the Internet of Things 2008 Conference*. Zurich, Switzerland. Springer Berlin Heidelberg, 308-324.

Vermesan, O., Friess, P., Guillemin, P., Gusmeroli, S., Sundmaeker, H., Bassi, A., Jubert, I.S., Mazura, M., Harrison, M., Eisenhauer, M. and Doody, P. (2011) Internet of things strategic research roadmap. *Internet of Things-Global Technological and Societal Trends*, *1*, 9–52.

Viglas, S.D. and Naughton, J.F. (2002, June) Rate-based query optimization for streaming information sources IN: *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, June 3. ACM, 37–48.

Viglas, S., Naughton, J. and Burger, J. (2003) Maximizing the output rate of multi-way join queries over streaming information sources IN*: Proceedings of the 29th International Conference on Very Large Data Bases*, Sep 9. Berlin, Germany. VLDB Endowment, 285-296.

Vila, L. (1994) A survey on temporal reasoning in artificial intelligence. *AI Communications*, 7(1), 4–28.

Vitter, J. (1999) External memory algorithms and data structures IN*:* James, M. and Jeffrey Scott, V. (eds.*) External memory algorithms*. American Mathematical Society.

Volz, R., Staab, S. and Motik, B. (2005) Incrementally maintaining materializations of ontologies stored in logic databases IN: *Journal on Data Semantics II*. Berlin/Heidelberg: Springer, 1–34.

Walavalkar, O., Joshi, A., Finin, T. and Yesha, Y. (2008) Streaming knowledge bases IN: *International Workshop on Scalable Semantic Web Knowledge Base Systems*, Oct.

Wei, C. and Li, Y. (2011, September) Design of energy consumption monitoring and energy-saving management system of intelligent building based on the Internet of things. In *2011 International Conference on Electronics, Communications and Control (ICECC)*, IEEE, 3650–3652.

Whitehouse, K., Zhao, F. and Liu, J. (2006) Semantic streams: A framework for composable semantic interpretation of sensor data IN: R¨omer, K., Karl, H. and Mattern, F. (eds.) *EWSN 2006, LNCS*, vol. 3868. Heidelberg: Springer, 5–20.

Wilschut, A. and Apers, P. (1993) Dataflow query execution in a parallel main memory environment. *Distributed Parallel Databases*, 1(1), 103–128.

Wingerath, W., Gessert, F., Friedrich, S. and Ritter, N. (2016) Real-time stream processing for Big Data. *it-Information Technology*, 58(4), 186–194.

Woods, W. (1975) What's in a Link: Foundations for Semantic Networks, IN: Bobrow, D.G. and Collins, A.M. (eds.), *Representation and Understanding: Studies in Cognitive Science*, Academic Press, New York, 35–82.

Wright, I. and Marshall, J. (2003) The execution kernel of RC++: RETE*, a faster RETE with TREAT as a special case. *International Journal of Intelligent Games and Simulation*, 2(1), 36–48.

Wu, E., Diao, Y., and Rizvi, S. (2006) High-performance complex event processing over streams. In SIGMOD '06: Proceedings of the 2006 ACM SIGMOD international conference on Management of data. ACM, New York, NY, USA, 407–418.

Wygant, R.M. (1989) CLIPS—A powerful development and delivery expert system tool. *Computers & industrial engineering*, 17(1-4), 546–549.

Xing, Y., Zdonik, S. and Hwang, J. (2005) Dynamic load distribution in the borealis stream processor IN*: Proceedings* of *the International Conference on Data Engineering (ICDE'05)*, Apr 5. IEEE, 791-802.

Yu, P., Chen, M.S., Wolf, J. and Turek, J. (1993) Parallel query processing. *Advanced Database Systems*, 229–258.

Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., Franklin, M.J., Shenker, S. and Stoica, I. (2012, April) Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation* (pp. 2-2). USENIX Association.

Zaharia, M., Das, T., Li, H., Hunter, T., Shenker, S. and Stoica, I. (2013, November) Discretized streams: Fault-tolerant streaming computation at scale. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles.* ACM, 423–438.

Zeitler, E. and Risch, T. (2011) Massive Scale-out of Expensive Continuous Queries. *Proceedings of the VLDB Endowment, 4*(11).

Zhang, Y., Duc, P., Corcho, O. and Calbimonte, J. (2012) SRBench: A streaming RDF/SPARQL benchmark IN: *The Semantic Web–ISWC 2012*. Heidelberg: Springer, 641–657.