Event Processing Using Semantic Web Technologies

Mikko J. Rinne
Event Processing Using Semantic Web Technologies

Mikko J. Rinne

A doctoral dissertation completed for the degree of Doctor of Science (Technology) to be defended, with the permission of the Aalto University School of Science, at a public examination held at the lecture hall T2 of the school on 1 September 2017 at 12 noon.

Aalto University
School of Science
Department of Computer Science
Semantic Computing Research Group
Supervising professor
Professor Heikki Saikkonen, Aalto University, Finland

Thesis advisors
Dr. Seppo Törmä, Aalto University, Finland
Dr. Esko Nuutila, Aalto University, Finland

Preliminary examiners
Professor Opher Etzion, Yezreel Valley College, Israel
Dr. Ora Lassila, Pegasystems Inc., USA

Opponent
Professor Alessandro Margara, Politecnico di Milano, Italy
Abstract

The massive increase in the availability of event information originating from networked sensors and transactions has lead to a situation, where fast stream processing of large quantities of data is becoming a critical competitive advantage for a wide variety of domains in both business and life in general such as retail, investment market, farming, emergency response and health services. The increasing need to fuse data streams from a variety of sources emphasises the importance of managing heterogeneous ontological approaches.

The founding principle of complex event processing is to abstract simple lower-level measurements and observations into more tangible higher-level conclusions, typically smaller in number and therefore easier to archive, present to a human reader or provide as input to the next processing layer. In this study methods for layered processing of event patterns using Semantic Web technologies have been developed.

SPARQL Query and Update constructs have been used to build query networks, capable of functioning as event processing applications without extensions to the original syntax of the language. By means of a new ontology design pattern representations for, e.g., composite and complex events have been constructed. Examples of all the different types of event processing agents found in literature have been implemented and tested for performance on a platform developed for this study. The approach has been tested also with a practical application related to pharmaceutical manufacturing. The same application has been implemented on an event processing platform from outside the Semantic Web domain for a performance comparison.

Entailment regimes for rule-based reasoning have been built using query networks similar to the ones used for building applications, making it easier for an application developer to improve efficiency through customisation and optimisation of the sets of rules. Good compliance of the implemented regimes has been tested using the official test set.

Based on the experiences proposals for future versions of SPARQL to better support event processing applications are made. The program code, query networks, test data and the related documentation are made available with an open license to support independent verifications of the tests and further development of the code and query networks. Based on the implemented query networks and tests it is shown that RDF and SPARQL offer sufficient capabilities to function as the basis of layered complex event processing applications.

Keywords complex event processing, data stream processing, SPARQL, RDF, Semantic Web
Tekijä
Mikko J. Rinne

Väitöskirjan nimi
Tapahtumankäsittely semanttisen webin teknologioiden käyttäen

Julkaisija
Perustieteiden korkeakoulu

Yksikkö
Tietotekniikan laitos

Sarja
Aalto University publication series DOCTORAL DISSERTATIONS 125/2017

Tutkimusala
Tapahtumankäsittely

Käsikirjoituksen pvm
20.06.2017
Väitöspäivä
01.09.2017

Julkaisuluvan myöntämispäivä
23.05.2017
Kieli
Englanti

Monografia
Artikkeliväitöskirja
Esseeväitöskirja

Tiivistelmä
Verkkoon kytkeytyvien antureiden tuottamien ja muun tapahtumia rekisteröivän tiedon valta
kasvu on johtanut tilanteeseen, jossa suurten tietomärien nopeasta vuoroprosessoinnista on tulossa
kriittinen kilpailuettu sekä liike- että muun elämän alueille kuten kauppaan, sijoituskärkirkoille,
maanviljelykseen, pelastustoimeen ja terveyspalveluihin. Kasvava tarve yhdistellä tietovirtoja
erityyppisistä lähteistä korostaa erilaisten käsitteistölisten lähestymistapojen hallinnan tärkeyttä.

Kompleksitaapahmutimen käsitteilyn pääperiaate on tiivistää yksinkertaiset aleman tason
mittaukset ja havainnot helpommin ymmärtettäviksi korkeamman tason päätelmiksi, joita
tyypillisesti on lukumääräisesti vähemmän ja jotka siksi ovat helpompia arkistoida, esittää
ihmiselle tai tarjota syötteenä seuraavalle käsitteilykerrokselle. Tässä tutkimuksessa on kehitetty
menetelmiä tapahtumasarjojen kerrokselliseen käsitteilyyn semanttisen webin teknologioiden

SPARQL kyselykielen ja päivityssääntöjen avulla on rakennettu sääntöverkkoko, jotka kykenevät
toimimaan tapahtumankäsittelyohjelmistoa ilman laajempia kielien alkuperäiseen syntaksisi.
Uuden ontologiakomponentin avulla on rakennettu esitysmuodot esimerkkiksi yhdistelmä- ja

kompleksitaapahmusteelle. Esimerkkitapaukset kaikista kirjallisuudesta löydetystä
tapahtumankäsittelyn modulilityypeistä on toteutettu ja niiden suorituskyky testattu tutkimusta
varten kehitettyllä ohjelmistoalustalla. Lähestymistapa on testattu myös lääkevalmistukseen
liittyvällä käytännön sovelluksella. Sama sovellus on toteutettu ilman semanttisen webin
esitystapoja toimivalla tapahtumankäsittelyohjelmistolla suorituskykyvertailua varten.

Sääntöpoljaisen pääätelyn käyttämää säännöstöjä on toteutettu käyttäen samanlaisia
kyselyverkkokoja kuin sovelluksissa, mikä helpottaa sovelluskehittäjän työtä säännöstöjen
raätälöinnissä ja optimoinnissa. Kehitettyjen säännöstöjen hyvä yhteensopivuus on testattu
virallista testijoukkoa käyttäen.

Kokemusten pohjalta tehdään ehdotuksia SPARQL-kielen jatkokehitykseen, jotta tulevat versiot
paremmin tukisivat tapahtumankäsittelysovelluksia. Ohjelmakoodei, kyselyverkot, testidata ja
kokeisii liittyvää dokumentaatio ovat tarjolla internetissä avoimella lisenssillä, jollakin kokeet ovat
riippumattomasti toistettavissa ja koodit ja kyselyverkot jatkokehittävissä. Esitetyjen
sääntöverkkojen ja testien perusteella työssä osoitetaan, että RDF ja SPARQL sisältävät riittävät
ominaisuuksia monimuotoisen terveyksellisen tapahtumankäsittelyn sovellusten pohjana

Avainsanat
kompleksitapahtumien kasittely, vuoroprosessointi, SPARQL, RDF, semanttinen web

ISBN (painettu)
978-952-60-7510-5

ISBN (pdf)
978-952-60-7509-9

ISSN-L
1799-4934

ISSN (painettu)
1799-4934

ISSN (pdf)
1799-4942

Julkaisupaikka
Helsinki

Painopaikka
Helsinki

Vuosi
2017

Sivumäärä
176

urn
Little did I know in 2010, when the opportunity to dedicate some time for research presented itself, to what kind of places that journey would take me. I knew I wanted to catch up with the 14 years of computer science that had passed after my graduation, I had a list of topics I was interested in and absolutely no publishing history on any of them. I was registered as a postgraduate student in communications and networking since 1997, which did not look too promising either. I knew I wanted a collection of publications and a team to write them with, but I wasn’t in contact with anyone doing research in any of those domains. Despite the challenges I sent emails to professors in related areas, quit my job and registered for all suitable postgraduate courses, trusting that practicalities would eventually sort themselves out.

The first person to thank at the university is professor Heikki Saikko-nen for taking the leap of faith and believing that despite the time in the industry and a CV of patents instead of scientific publications I could still be a useful member of the community. Most of the ideas in this work follow the thoughts of Dr. Seppo Törmä, my advisor, who always had the time to sit down and discuss any issues discovered during the journey. Dr. Esko Nuutila, my other advisor, took it upon himself to work on the software platform that all of my work here is based on. The shape that this dissertation eventually took is mostly due to Seppo and Esko disagreeing with many aspects of the previous proposals for RDF stream processing. And I agreed with them.

I was first planned to work on “the temporal aspects of event processing using Semantic Web methods”, but it soon became clear that there were plenty of other pieces missing from event processing with Semantic Web methods in general before the specific domain of temporality could be reached. The concept of time remains an essential building block of my
work, but it could only be addressed after a larger domain of event-related mechanisms, which had to be developed in order to be able to experiment with a working system.

I am very grateful to my other co-authors Haris Abdullah, Eva Blomqvist, Robin Keskiärkkä and Monika Solanki for the opportunities to combine our expertise in creating something new, and for the great collaboration I have enjoyed in all of these projects. I would also like to acknowledge the warm welcome I received in the RDF stream processing research community, and the inspiring discussions with numerous researchers in the domain.

I probably wouldn't even have thought about postgraduate studies without the encouragement and support of Timo Ali-Vehmas, Heikki Huomo, Yrjö Neuvo, Heikki Ahava, Harri Honkasalo and Kari Pehkonen, my superiors and mentors at Nokia. Discussions with professor Sumi Helal from the University of Florida have been a great inspiration both during and after his visit to EIT Helsinki.

I wish to take the opportunity to thank my parents, Juhani and Eliisa Rinne, for all the love, support and endless faith they have provided during my first 46 years. I thank Woonhee for all the support throughout my academic efforts and our children, Ennu and Leo, for giving me faith for the future and preventing me from getting too deep into semantics.

Espoo, Finland, June 20, 2017,

Mikko J. Rinne
Contents

Preface 1

Contents 3

List of Publications 5

Author’s Contribution 7

List of Acronyms 8

1. Introduction 11

2. Literature review 17
   2.1 Events ........................................ 17
      2.1.1 Event definition alternatives .................. 19
   2.2 RDF ........................................ 19
   2.3 Non-RDF representations for event objects ............ 20
      2.3.1 Tabular representations ..................... 20
      2.3.2 Tree representations ....................... 21
   2.4 Event-related ontologies .......................... 22
   2.5 Event streams .................................. 23
      2.5.1 Timestamps on streamed entities ............... 24
   2.6 Event stream processing background .................. 24
      2.6.1 Complex event processing platforms .......... 25
      2.6.2 Data stream management systems ............. 26
      2.6.3 Publish subscribe interaction paradigm ....... 29
   2.7 Stream reasoning ................................ 29
   2.8 Benchmarking of stream processing platforms .......... 30
      2.8.1 Published performance results ............... 30

3. Events, states and background knowledge 33
3.1 Event objects ........................................ 33
  3.1.1 Time in events ................................... 35
3.2 States .................................................. 37
3.3 Background knowledge ............................... 38
3.4 Example of events, states and background knowledge .... 39

4. RDF-encoding of structured objects .................... 41
  4.1 Event Processing ODP ............................... 41

5. Streaming RDF ........................................... 45
  5.1 Solutions for timestamps ............................ 46
  5.2 Heterogeneous events in an RDF stream ............. 48
    5.2.1 Issues with streaming Turtle .................. 48
    5.2.2 Streaming a dataset of graphs ................. 49

6. Event stream processing ................................ 51
  6.1 Mapping of EPN elements ............................ 52
  6.2 Defining event patterns ............................. 53
  6.3 Event processing with networks of SPARQL .......... 54
    6.3.1 Advances in event processing ................. 56
    6.3.2 Challenges of SPARQL in event processing .... 57
  6.4 Challenges of Data Stream Management Systems (DSMS) 59
  6.5 Stream descriptions ................................ 61
  6.6 Memory management .................................. 62

7. Reasoning on events .................................... 65

8. Performance ............................................. 69
  8.1 Summary of INSTANS performance tests ............. 71
  8.2 INSTANS throughput comparison .................... 75
  8.3 Observations on the performance of INSTANS ....... 78

9. Discussion ................................................ 81

References ............................................... 89

Publications ............................................ 101
List of Publications

This thesis consists of an overview and of the following publications which are referred to in the text by their Roman numerals.


Author’s Contribution

Publication I: “Processing Heterogeneous RDF Events with Standing SPARQL Update Rules”

The first author planned, coded, executed and documented the experiment. Platform coding was carried out by the second author.

Publication II: “Event Processing in RDF”

The first author created, tested and documented the example scenario and the event structures. The queries were written jointly by the authors. The ontology design pattern was prepared by the second author.

Publication III: “Constructing Event Processing Systems of Layered and Heterogeneous Events with SPARQL”

The first author prepared everything apart from INSTANS platform coding.

Publication IV: “RFID-based logistics monitoring with semantics-driven event processing”

The first author planned, created, carried out and documented the experiment. The test set was specified jointly by the first and second authors. The example use case and datasets were provided by the second author. The third author provided platform coding.
Publication V: “User-Configurable Semantic Data Stream Reasoning Using SPARQL Update”

The first author prepared everything apart from INSTANS platform coding.

List of Acronyms

ABox  Assertional Box
API  Application Programming Interface
BSBM  Berlin SPARQL BenchMark
C-SPARQL  Continuous SPARQL
CEP  Complex Event Processing
CL  Common Lisp
CQELS  Continuous Query Evaluation over Linked Streams
CQL  Continuous Query Language
CSV  Comma Separated Values
CV  Curriculum Vitae
DBMS  Database Management Systems
DEBS  Distributed and Event-Based Systems
DL  Description Logics
DOLCE  Descriptive Ontology for Linguistic and Cognitive Engineering
DSMS  Data Stream Management Systems
DUL  DOLCE Ultra Light
ECA  Event-Condition-Action
EIT  European Institute of Innovation & Technology
EP-SPARQL  Event Processing SPARQL
EPA  Event Processing Agent
EPC  Electronic Product Code
EPL Event Processing Language
EPN Event Processing Network
eps events per second
ETALIS Event TrAnsaction Logic Inference System
EXI Efficient XML Interchange
FIFO First In First Out
GPS Global Positioning System
HTTP HyperText Transfer Protocol
HW Hardware
IETF Internet Engineering Task Force
IFP Information Flow Processing
INSTANS INcremental engine for STANding Sparql
IRI Internationalised Resource Identifier
JoDS Journal on Data Semantics
JSON-LD JavaScript Object Notation for Linked Data
JSON JavaScript Object Notation
LEAPS Lazy Evaluation Algorithm for Production Systems
LNCS Lecture Notes in Computer Science
LODE Linked Open Descriptions of Events
LUBM Lehigh University Benchmark
ODBASE Ontologies, DataBases and Applications of Semantics
ODP Ontology Design Pattern
OS Operating System
OSI Open Systems Interconnection
OWL Web Ontology Language
QL Query Language
R2S Relation-To-Stream
RDFS RDF Schema
RDF Resource Description Framework
RESL RDF Event Subscription Language
RFC Request for Comments
RFID Radio-Frequency IDentification
RSP RDF Stream Processing
S2R Stream-To-Relation
SBCL SteelBank Common Lisp
SFP Semantic Flow Processing
SPARQL SPARQL Protocol and RDF Query Language, a recursive acronym
SQL Structured Query Language
SSN Semantic Sensor Network
SW Software
SWCLOS Semantic Web Common Lisp Object System
TBox Terminological Box
TESLA Trio-based Event Specification Language
TRDF Temporal RDF
URI Uniform Resource Identifier
W3C World Wide Web Consortium
WOP Workshop on Ontology and semantic web Patterns
WQL Wilbur Query Language
XML eXtensible Markup Language
XSD XML Schema Definition
1. Introduction

“She knows herself to be at the mercy of events, and she knows by now that events have no mercy.”

– Margaret Atwood, The Blind Assassin, 2000

Human languages tend to be gravely inaccurate on the definition of “an event”. They make no distinction between something instantaneous such as the closing of a door or the activation of a fire alarm, and something longer-lasting [8], such as the Great Fire of Rome in 64, the Woodstock rock festival in 1969 or even the Second World War from 1939 to 1945, all of which can be referred to as “historical” or “memorable” events. In this body of work, an event refers to the first and instantaneous kind.

With or without our knowledge or consent, events take place: raindrops fall on the ground and lightning bolts strike, sometimes with dire consequences. While humans rely on senses, a computer can be informed of an event of the material world through sensors. In addition to mechanical and electrical sensors such as water level meters and voltage probes, humans are increasingly providing sensory input, e.g., when the spreading of infectious diseases is tracked from popular internet searches [55] or microblogging entries [35]. Often a combination of the output from different types of sensors is fused to deduce that a real-world event has taken place. When a thermal sensor goes over a temperature threshold and a smoke detector activates in the same space, a surveillance computer can generate a “fire alarm”, a complex event object [89] derived from those two measurements.

The proliferation of networked sensors [6] in our environment is rapidly extending our capability to monitor our surroundings all the way from our personal health and homes to global weather phenomena. Our mobile devices can measure our location and provide an estimate of our context,
Introduction

cars are fitted with an average of 60-100 sensors\textsuperscript{2} and remote surveillance and control of heating are increasingly common in new homes. As “physiological” and “safety” are the first two levels of Maslow’s hierarchy of needs\textsuperscript{91} it is no surprise that relevance in pilot applications of event processing is being sought from monitoring of crops\textsuperscript{123}, stock markets\textsuperscript{2,128}, natural disasters\textsuperscript{72} and traffic\textsuperscript{13,50,84}.

While the increasing quantity of streamed data is a massive enabler, it also presents huge challenges. A digital photographer may have faced the problem of quickly accessing the truly great shots of a lifetime, while also having to archive all the 120 consecutive shots of one sunset at Grand Canyon for the case of a sudden desire to create a timelapse 40 years later. The sheer quantity of data our sensors generate is such that long-term archiving of every measurement is seldom feasible. At the same time the usefulness of real-time data in, e.g., farming, traffic and business decays quickly. Event processing aims to keep up with Big Data\textsuperscript{3} on the fly, primarily for the purposes of preventive or fast reactive measures.

An important part of this analysis is the hierarchical abstraction of large quantities of low-level bits and pieces of information into more tangible, higher-level events, which are easier to process and more feasible to archive. The term complex event processing\textsuperscript{88} has been used for this layered approach.

In addition to the quantity of data, the diversity of platforms presents a significant and increasing challenge. Smart cities are built, owned and operated by multiple actors sourcing their equipment from multiple vendors. A market survey from 2016\textsuperscript{4} lists 33 currently supported systems for complex event processing. Even though there are common formats for exchanging data, there are at present neither means to define global concepts nor common ways for configuration within that set of tools.

The World-Wide Web\textsuperscript{23} has become a ubiquitous repository of documents, but it is also emerging as a real-time information platform. The Semantic Web\textsuperscript{24}, also referred to as the Web of Data\textsuperscript{5}, is a framework for describing things. It provides a natural platform for publishing sensor data\textsuperscript{85}. The principles of using a globally routable Uniform Resource

\textsuperscript{2}http://www.automotivesensors2015.com/ [Accessed Feb 12th 2016]
\textsuperscript{4}http://www.complexevents.com/2016/05/12/cep-tooling-market-survey-2016/ [Accessed 9th Aug 2016]
\textsuperscript{5}https://www.w3.org/DesignIssues/Semantic.html [Accessed Feb 12th 2016]
Introduction

Identifier (URI) [25] for things and globally accessible ontologies\(^6\) for modelling the relationships between them can bring order – or at least greatly improved structure – to an otherwise chaotic environment. The Semantic Web is built on an open world assumption\(^7\) and it provides tools to cope with overlapping definitions from multiple sources. The capability to cope with heterogeneous sources of event data extends also to background information, where Linked Open Data\(^8\) interfaces naturally with event data encoded with Semantic Web technologies. This combination yields extreme flexibility for supporting loosely coupled heterogeneous actors for event data production, consumption and static background information access, and paves the way for a higher level of automation.

Another important enabler embedded in the Semantic Web technologies is the support for reasoning [105]. In the context of event processing a stream of events can be complemented with data synthesised using predefined rules and axioms. With reasoning the system can have a better understanding of the events being processed. Missing event data can be filled in, if it can be deduced from existing data and a set of rules. Errors can be automatically detected, if data contradicts with the rules. Reasoning can also facilitate integration in a multi-vendor environment and support rule-based automation of organising event-producing sensors.

This dissertation investigates the use of Semantic Web tools RDF [140] (Resource Description Framework, as the metamodel) and SPARQL [136] query language and update rules (as the rule framework and application programming language) in complex event processing tasks. Related experiments are performed on INSTANS\(^9\), a platform for continuous processing of RDF-encoded data streams using networks of rules encoded in SPARQL.

There are also challenges involved in applying the Semantic Web to event processing. As long as event streams are produced in homogeneous, closed and controlled environments, defining formal ontologies to describe attributes and their relations may seem like design overhead in the system specification phase, and tagging those attributes with references to

---

\(^6\)https://www.w3.org/standards/semanticweb/ontology [Accessed Sep 30th 2016]

\(^7\)unavailability of a given assertion or fact does not imply it to be either true or false [101]

\(^8\)http://linkeddata.org/ [Accessed Mar 24th 2016]

the defined ontologies may seem like transmission overhead during operation. The RDF data model does not natively incorporate a concept for a document object (as in, e.g., eXtensible Markup Language (XML) [133]). Therefore the delimitation of events in a stream requires a solution, e.g., by using a separate named graph per event. Finally, even though the free-form graph structure of RDF offers high flexibility for variations in the format of an event object, it also means that no simplifying assumptions on the event processing platform can be made for parsing and processing a received event object. In other event processing environments the schemas of permissible event objects either may or have to be defined a priori, allowing the platform to optimise the parsing of received event objects to programming language objects and their subsequent processing.

When this work started, the primary method for RDF stream processing was the repeated execution of SPARQL queries over time windows extracted from a stream of timestamped RDF triples. Could RDF also be used for the encoding and SPARQL for the processing of streams of multiple different types of structured multi-attribute events with optional elements? How could layered abstractions, i.e. either referencing or encapsulating events within events, be represented, queried and processed? How should these structured events be encoded to support consumers starting to receive the stream from an arbitrary point in time, without ambiguity of event borders? How should timestamps be encoded to have scope over the whole event, and could there be multiple timestamps to reflect different points of measurement during the lifetime of an event object? Could SPARQL also be used for hierarchical processing of event abstractions? How about reasoning? What else than SPARQL would be required to define an event processing application? How much could be achieved without breaking compatibility with existing RDF and SPARQL specifications and tools? Would some extensions be necessary to meet these goals? If not, which optional extensions would be the most important ones?

The main contribution of this work is the development and demonstration of methods for layered processing of patterns of structured and heterogeneous events using Semantic Web technologies. The attached publications show how asynchronously operating networks of specification-compliant SPARQL Query and Update rules can be used to build event processing applications. The concept is demonstrated and the performance compared to another system in the RDF stream processing domain.
in Publication I. A new *Ontology Design Pattern (ODP)*\(^{10}\) is introduced to incorporate the key definitions of complex event processing, such as composite and complex events, into the RDF domain in Publication II. Publication III extends on Publication I by demonstrating how the different types of *Event Processing Agents (EPA)* appearing in related literature [53, 123] can be encoded as SPARQL networks and interconnected to form a larger *Event Processing Network (EPN)*. In Publication IV a real-life monitoring task from the pharmaceutical industry is implemented both on INSTANS as well as a commercial event processing product from outside the Semantic Web domain, comparing the two qualitatively from the perspective of event processing application implementation and quantitatively on throughput performance. Having demonstrated the applicability of RDF and SPARQL in this problem domain, the beneficial features of Semantic Web technologies are leveraged through the construction of a set of entailment regimes for rule-based reasoning. In Publication V multiple different reasoning frameworks are implemented as user-configurable libraries with SPARQL, assessing their conformance with the SPARQL entailment test suite and comparing their performance on INSTANS with two non-streaming platforms using a well-known benchmark. The entailment regimes are built as libraries of SPARQL query networks, open for customisation and optimisation. Compliance of the implemented regimes is verified using the official test set. Based on the experiences proposals for future versions of SPARQL to better support event processing applications are made. The program code, query networks, test data and the related documentation are made available with an open license to support independent verifications of the tests and further development of the code and query networks.

\(^{10}\)http://ontologydesignpatterns.org/wiki/Main_Page

[Accessed Sep 30th 2016]
2. Literature review

“In great literature, I become a thousand different men but still remain myself.”

– C.S. Lewis, An Experiment in Criticism, 1961

This review of background information introduces the reader to the domain of the dissertation by compiling references to earlier work on the processing of streams of temporal information, both with and without the context of the Semantic Web. It presents the key concepts and definitions of different categories of events, which are used in the modelling of event processing tasks in the included publications. Knowledge of the history of stream processing and event processing platforms is necessary to arrive at the design choices made with the new platform to overcome some of the limitations of prior approaches.

This chapter progresses in the same order of topics as the subsequent chapters summarising the contribution of the publications included in the dissertation: from the representation and classification of temporal information to methods of encoding, creation of streams, stream processing, reasoning and measuring performance.

2.1 Events

The Event Processing Glossary [89] defines an event as “anything that happens, or is contemplated as happening”. This broad definition ties events to the material world. In Publication II the term event object, also found in [89], is used to denote a computer-encoded representation of an event.

From the categories of events described in [89], the following are the definitions for the ones mainly referenced in the context of this dissertation:
• **Composite event:** created by combining a set of other simple or complex events, always including the member events from which it is derived.

• **Complex event:** summarises, represents or denotes a set of other events.

• **Simple event:** not viewed as summarising, representing or denoting other events (the complement of a complex event).

In [53] the structure of an event is further split between a header and optional payload. Payload is shown as a list of typed attributes (string, numeric types, boolean, date, time, location, reference to other event) with no defined semantic meaning in the current EPN.

The convention of assigning single timestamps to event objects – also complex events – is followed by Zimmer and Unland in [145]. They use the term *initiator* for the oldest component event and the term *terminator* for the youngest, and set the occurrence time of the complex event equal to the occurrence time of its terminator\(^1\).

Moving from events to intervals, time-based validity relations of two intervals are covered by the thirteen relations of *Allen’s interval algebra* [8]. Interval semantics are also used in, e.g., [45].

The term *background knowledge* is used to describe information, which contains no associated time of validity [9, 15, 128]. The term “static data” is also sometimes used to describe such information [75], but as there is no guarantee that the data would not change – only the lack of information on when that would happen – “semi-static” is hereby considered a more accurate description of persistence of such data.

The concepts of Terminological Box (TBox) and Assertional Box (ABox) are used in the context of the Description Logics (DL) family of formal knowledge representation languages [29, 62]. The TBox contains assertions on concepts and the ABox contains assertions on individual objects, typically related to an individual being an instance of a certain concept. Taking an example from the Lehigh University Benchmark (LUBM) [66] used in Publication V, a TBox ontology can state that *AssistantProfessor* is a class and a subclass of *Professor*. From the LUBM ABox data we learn that *AssistantProfessor\(^0\)* (a computer-generated entity) is an instance of *AssistantProfessor*. Using the TBox information we can reason that *AssistantProfessor\(^0\)* is also a *Professor*.

\(^1\)Several initiators and terminators are allowed.
2.1.1 Event definition alternatives

The concepts and definitions in this domain have not fully stabilised. The term “fact” has been used as a synonym for what is called “state” in Section 3.2 of this dissertation [60, 75]. More commonly a “fact” would be viewed as a semantic term, possibly represented by a state, but not as a synonym.

“Fact” is also overloaded in the domain of interest, because an RDF triple (ref. Section 2.2) without time annotation (ref. Section 2.5.1), which belongs to background knowledge (ref. Section 3.3) under the definitions of this dissertation, has also traditionally in the context of RDF been called a “fact”\(^2\).

Another approach to the relation of time and event objects is to attach two timestamps to every event object and not incorpore a separate definition for a time interval. In [9] the term “temporal knowledge” is used to cover both events and intervals.

2.2 RDF

RDF is a framework for representing information on the Web [140]. As a core building block of the Semantic Web, it is directly compatible with the SPARQL query language. The core structure is a three-element directed graph, known as a triple. A triple consists of two nodes – a subject and an object – and a predicate, which connects the subject to the object. A triple can be expressed as:

\[
<s, p, o> \in (I \cup B) \times I \times (I \cup B \cup L)
\]  

(2.1)

where \(I\), \(B\) and \(L\) are sets of IRIs, blank nodes and literals, respectively. A set of RDF triples is known as an RDF graph \(G\). Multiple serialisation formats have been defined for storing and exchanging RDF data. Commonly used serialisations are RDF/XML [133], Turtle [22], NTriples [21] and JSON-LD [118]. Hierarchical information can be expressed as RDF by using intermediate subjects – often blank nodes [140] – to connect multiple levels of predicates and objects.

2.3 Non-RDF representations for event objects

As this study is based on Semantic Web methods, no detailed survey on other event object representations was carried out. Despite this, other event object representations appear in parts of the work, especially in Publication IV, where another event processor – Esper\(^3\) – was used for comparison. Common approaches for encoding event objects are tabular [42] and tree [5] formats. Their application to event object encoding is shortly discussed below.

2.3.1 Tabular representations

Tabular information is arranged in a two-dimensional table having columns and rows, where the number of columns is typically fixed and the number of rows is dynamic based on the size of the table. The label and format of each column is specified jointly for the whole table. If values for all columns do not have values on all rows, they can be left blank. Tabular formats are a typical way to encode database records or spreadsheet tables.

Database relations are unordered sets of records [42], whereas time-varying data streams are typically delivered in an approximate order of creation\(^4\). Tabular formats are widely used for encoding homogeneous data streams, where all rows (usually called tuples in the context of data stream processing [2]) contain the same attributes (some of which can be optional). Because observers of data streams may start receiving the stream at any point in time, column format specifications cannot be delivered only on the first line of transmission. The column format needs to be either repeated in the stream, or made accessible to recipients by other means. Tabular formats could be used to communicate event object information from active relational Database Management Systems (DBMS) [96], as long as rows from only one table need to be encoded. For managing objects extracted from multiple tables (to encode complex or composite event objects), another data structure on top of tabular would be needed. In practice such cases are handled with tree-format objects.

Comma Separated Values (CSV) has been a de facto standard for exchange of tabular information between, e.g., spreadsheet tools for a long


\(^4\)Variation in transmission delay from originating node to event processing platform may alter the order, which needs to be taken into account in processing.
time, but Internet Engineering Task Force (IETF) RFC4180 [113] from 2005 is the self-proclaimed first formal specification. CSV is used widely for stream processing outside the Semantic Web domain due to compactness, good human readability, tool support and ease of implementation. CSV works well for uniform content when the format is known a priori and distributed to recipients outband. Uniform CSV content can also be converted to RDF5.

A stream of lines of comma-separated attributes can also be used to transfer multiple pre-agreed event types, if the first column is agreed to specify the event type, determining the interpretation for the rest of the columns on the same line. This approach is compliant with the informative grammar of [113], which specifies the header row as optional and does not require rows to have an equal number of columns. The approach requires a custom outband specification of the event type specifiers and would not be commonly understood as CSV.

CSV is used as the output format in Publication IV and Publication V for result comparison due to easy ordering properties: As each event object is represented by one line of text, output can be sorted and compared using commonly available tools.

### 2.3.2 Tree representations

A tree is a graph, in which all nodes are reachable from a root node and which does not contain cycles [5]. Tree structures are used in a wide variety of computing tasks, and are well-suited for encoding hierarchical entities including composite event objects.

XML [134] has become ubiquitous on the Web. Each XML document contains one tree6, and XML documents are not originally defined as a stream. A typical way for streaming XML documents is to use outband delimitation, e.g., transmit each XML document in a separate HyperText Transfer Protocol (HTTP)7 request. Efficient XML Interchange (EXI) [139] streams are a newly specified way of sending XML event streams. XML supports global namespace definitions, although they are not fully compatible with the Web Ontology Language (OWL) ontologies used in RDF. XML document structure can be defined in the form of a schema8.

---

5https://w3c.github.io/csvw/csv2rdf/ [Accessed Sep 29th 2016]
6https://www.w3schools.com/xml/xml_tree.asp [Accessed Feb 28th 2017]
7https://www.w3.org/Protocols/ [Accessed Mar 14th 2016]
Schemas can assist in optimising the implementation of an event processing agent, but they could also be seen as restricting towards a heterogeneous event stream. As the use of an XML schema is not mandatory, it is observed as an optional feature, which may or may not be used and is therefore not restricting the suitability of XML for event object encoding. XML was used as the event format in the Esper environment in the experiments of Publication IV.

JavaScript Object Notation (JSON) has become a popular data interchange format, used for event encoding, e.g., in [36]. JavaScript Object Notation for Linked Data (JSON-LD) [118] bridges the gap between JSON and RDF. JSON-LD was not used in any of the included experiments, but it would be a potential way to convert RDF data for, e.g., Esper.

2.4 Event-related ontologies

The use of globally accessible ontologies for concepts connects the information in events to the global repositories of linked open data and the ontologies defined therein. The same conceptual linking, which takes place between well-prepared linked open data [73] is enabled for events. When event streams from multiple systems or vendors become available, the ability to manage flexible linking of concepts from heterogeneous sources will become increasingly valuable.

For expressing the different types of events from Section 2.1 in RDF, the definitions available in pre-existing event-related ontologies were reviewed in Publication II. The Event Ontology [11] – although based on music events – and the LODE ontology [12] are sufficiently generic to be usable also for event processing. An ontology for complex event processing [123] has been defined, but it is specific to the problem domain.

The event processing ODP defined in Publication II uses DOLCE Ultra Light (DUL) top-level ontology as formal basis, and aligns with the DUL-based Event-F ontology [112]. The Semantic Sensor Network
(SSN)\textsuperscript{15} [85] ontology is also based on DUL.

### 2.5 Event streams

An event stream can be primarily characterised as:

1. \textit{Infinite}: it has neither a beginning nor an end. [4, 11, 14, 129]

2. \textit{Ordered}: the order of events in the stream is significant. [28, 78, 131]

Other sources, e.g., the \textit{data stream model} [14], state additional characteristics, which are viewed here as optional properties of event streams:

1. Events in the stream arrive \textit{online} [14].

2. System has no control over the \textit{order of arrival} [14].

3. Events cannot be easily \textit{retrieved} after processing [14].

4. Events have \textit{timestamps} [2, 12, 14, 37, 119, 127].

5. \textit{Time-variant}: a new event may replace an old one [49].

6. Having \textit{non-decreasing timestamps} [15, 78].

Pre-RDF work on data stream processing [2] discusses \textit{tuples}, which are being timestamped by the stream processing platform upon arrival.

An important building block on the road to RDF streams is the \textit{annotation} of RDF. Gutierrez \textit{et al.} [67, 37] introduced \textit{temporal graphs} through the usage of timestamped triples. Lopes \textit{et al.} [86] widened the scope onto more general annotations of RDF on time, spatial and provenance. Both triple-specific and graph-specific annotations have been proposed. The primary methods of annotation are \textit{reification} [142], where original RDF triples are re-packaged as a triple per element (subject, predicate and object) so that the reified triple can have additional annotation fields, and \textit{quads}, where a fourth element is added to triples as a reference to annotations.

\textsuperscript{15}http://www.w3.org/2005/Incubator/ssn/ssnx/ssn [Accessed Oct 3rd 2016]
Since the definition of a graph does not require all nodes to be connected to a root node [5], separation of streamed heterogeneous events consisting of a variable number of triples requires a solution for punctuation [74, 129]. The TriG [26] specification defines a format for communicating a dataset of Turtle-serialised [22] RDF graphs.

2.5.1 Timestamps on streamed entities

Temporal RDF (TRDF) is defined in [15] as a stream of pairs of triples and timestamps:

\[
\begin{align*}
&(s_i, p_i, o_i, \tau_i) \\
&(s_{i+1}, p_{i+1}, o_{i+1}, \tau_{i+1})
\end{align*}
\]  

(2.2)

A TRDF stream consists of quads, where the fourth element is a single timestamp. Work on Event Processing SPARQL (EP-SPARQL) extended the RDF stream definition to two timestamps per triple, where \(t\) and \(t'\) are “time stamps denoting the boundaries of the time interval of the occurrence”. Tappolet and Bernstein [122] used versioned graphs, also with interval timestamps. The RDF Stream Processing (RSP) Community Group\(^{16}\) is working on a draft report\(^{17}\) and converging towards a model where RDF graphs can be annotated either with a single timestamp or an interval.

2.6 Event stream processing background

The first instance of computerised stream processing has been the programming of the first original First In First Out (FIFO) queue, and the serial processing of the elements placed in that queue, which is difficult to attribute to a person or a point in time. A good overview of Complex Event Processing (CEP) (Section 2.6.1) and Data Stream Management Systems (DSMS) (Section 2.6.2) is presented by Cugola and Margara [41]. They use Information Flow Processing (IFP) engine as a common term for CEP and DSMS. IFP systems processing RDF triples are further denoted Semantic Flow Processing (SFP) systems in [111].

\(^{16}\)https://www.w3.org/community/rsp/ [Accessed Mar 15th 2016]

2.6.1 Complex event processing platforms

Event processing originates from the fusion of rule-based expert systems (e.g., [57]) and their respective backends into active DBMS. The CODASYL Data Description Language and Programming Language Committees specified programming-language actions to be carried out based on input to a database, as described by Taylor and Frank in 1976 [124]. In 1983 Morgenstern [96] explains the principle of using constraint equations as declarative representations for expressing semantic constraints, such as integrity and consistency. He presents active databases, which instantly compute consequences of changes to data and automatically propagate the changes to actions, such as notification events to end-users. He also reviews both forward- and backward-chaining logic for re-establishing consistency in the database after a change in data. Active rule processing is also used by Dean and McDermott in [45]. McCarthy and Dayal (1989) [92] discuss the reception of external and temporal events from application or system processes, and the inference of complex events from primitive events based on the Event-Condition-Action (ECA) paradigm. They also interface their active database processing more tightly to applications: rule events can be defined by an application and results may contain requests to applications. Gatziu et al. [61] discuss complex events as “a set of events”, treating them as event patterns. They also support composite events described as “compositions of other events”. Rules in their system may fire in succession, but hierarchical processing of complex events is discussed more explicitly by Zimmer and Unland [145].

The latest wave of CEP systems was made popular by David Luckham [88]. The essence of CEP is hierarchical processing: simple lower-layer events and combinations thereof can be hierarchically abstracted into more tangible higher-layer events. Recognition of event patterns is often of importance in CEP applications.

A system for event processing can be described as an Event Processing Network (EPN) with the following elements (Publication III Figure 3 and [53]):

- **Event Processing Agent (EPA):** A software module, which processes events.

- **Event producer:** A source of events, which has no inputs in this EPN.
• **Event consumer**: A sink of events, which has no outputs in this EPN.

• **Event channel**: Implements the connections between event producers, EPAs and event consumers to form an EPN.

• **Global state elements**: Data available for EPAs.

• **Event Processing Network (EPN)**: Collection of event producers, EPAs and event consumers connected by event channels as well as the available global state elements.

Window processing is used in CEP applications, but more commonly as a temporary storage to hold incoming events for a certain period of time than to create event-independent time slices of the incoming stream for query processing. An example of this type of usage is shown for Esper in Publication IV, p. 4.

The INSTANS platform used in this study is based on the Rete algorithm by Charles Forgy [57], rooted in the rule processing of expert systems. Numerous variants of the original Rete have been created during the years, e.g., TREAT [94], Gator [68] and Lazy Evaluation Algorithm for Production Systems (LEAPS) [18], to name a few influential ones. In the context of RDF and SPARQL the Rete approach has been used in stream processing [64, 78] as well as linked data traversal [95].

Even though Structured Query Language (SQL) presents a common heritage in the query language DNA of the CEP platforms outside the RDF domain, the syntaxes are at the time of writing proprietary without any dominant specification. As two examples, Esper uses **Event Processing Language (EPL)** and WSO2\(^\text{18}\) uses **Siddhi**. **Trio-based Event Specification LAnguage (TESLA)** [39] is a formally defined event specification language, implemented on the **T-Rex** [40] middleware.

Characteristics of CEP can be found in some stream processors typically by offering filters, which recognise event patterns [9, 31].

### 2.6.2 Data stream management systems

The history with stream processing databases traces back to 1992 [127], when Terry *et al.* investigated the efficiency of continuous querying of in-\(^\text{18}\)http://wso2.com/products/complex-event-processor/ [Accessed Sep 18th 2016]
coming data on an add-only database instead of running periodic queries over the whole body of data. Their approach demonstrates the main differences of stream processing compared to traditional database access:

- In traditional database processing different queries are executed against a static body of data. Once all the data has been processed for a given query, the answer set is complete.

- In stream processing static queries are continuously processed or repeatedly executed against a (possibly infinite) stream of data. A set of answers can be considered correct for the processed segment of the data stream, but not complete.

The approach of [127] gave rise to a host of stream processing platforms, collectively referred to by, e.g., Babcock et al. [14] as Data Stream Management Systems (DSMS). Often-referenced examples include Aurora [2], Borealis [1] (a descendant) and STREAM [11]. The Continuous Query Language (CQL) [11] supports the window processing, which is characteristic for DSMS:

- **Stream-to-relation operators**, which separate (time-bound) segments of finite length from the stream.

- **Relation-to-relation operators**, which carry out relational database queries, typically as in SQL.

- **Relation-to-stream operators**, which emit the query answers as a stream.

Stream windows are an integral part of many RDF stream processing language extensions and engines such as Streaming SPARQL [28], Continuous SPARQL (C-SPARQL) [15, 16, 17], SPARQLStream / MorphStreams [30], Continuous Query Evaluation over Linked Streams (CQELS) [81], EP-SPARQL [9] and SparkWave [78]. Depending on the implementation, windows can be based on either time or a constant number of triples. The amount of overlap between consecutive windows is set as a parameter. Non-overlapping windows are often referred to as **tumbling** [11, 17, 33, 47, 58, 117]. Data stream processing is best suited
for producing aggregate values such as minimum, maximum, average or sum. Typical applications are the cleaning of noisy sensor data or the tracking of statistical trends over periods of time. RDF stream processing outside the context of time-variant streams and events has been discussed, e.g., in [83].

Outside the RDF and Semantic Web domains distributed stream processing has gained a lot of traction in the online big data analytics domain. An abundance of tools has emerged [69] with companies like Twitter (Apache Storm19, Heron20) [58], Microsoft (Trill21 [33, 34].NET library and Quill22 powering the Azure23 cloud analytics), Amazon (Kinesis24 cloud analytics), Google (Cloud Dataflow25), LinkedIn (Apache Samza26 [76] with Apache Kafka27 publish-subscribe messaging), EBay (Pulsar28), SAP (SAP Event Stream Processor29) and DataTorrent (Apache Apex30) actively involved in tool development. These projects also have an excellent track record in releasing their tools as open source (Storm, Heron, Samza, Kafka, Pulsar and Apex are available) for use and further development by the community. Viable open source competitors with origins in the academia are, e.g., the Apache Spark31 [114] engine originally developed at the University of California for large-scale data processing with a built-in streaming library and Apache Flink32 [31] with roots in the German university project Stratosphere33. Typically these tools only support programming language constructs instead of higher-level declarative query languages, but bridges to higher-level dataflow languages have been proposed, e.g., PipeFlow [110] for Storm and Spark.

2.6.3 Publish subscribe interaction paradigm

As another angle towards stream processing, the publish / subscribe (often referred to as “pubsub”) software architecture and interaction paradigm [54] emphasises a distributed system model, which decouples publishers (data object producers) from subscribers (data object consumers). Pubsub systems are rooted in the generative communication model emerging in late 1980s [32], with the name “publish / subscribe” gaining wider use in the early 1990s [99]. A typical addressing method in pubsub is topic-based subscription (e.g., [99]), where subscribers register their topics of interest and are thereafter asynchronously informed of new data objects published under their registered topics. When a system enables subscriptions through registration of more comprehensive persistent queries, the result is a distributed event stream processing system. The pubsub paradigm does not preclude the use of DSMS-like time windows, but it is not common, since pubsub applications do not typically target statistical processing of aggregate values. The pubsub paradigm can be employed as a single-layer architecture, i.e. it does not require a layered CEP system, but pubsub can also be implemented as the interaction model used between EPAs and EPNs, as events are published to event channels, which individual EPAs and EPNs subscribe to.

Examples of publish / subscribe systems using RDF data are JTangPS [115] (using a proprietary RDF Event Subscription Language (RESL)), Smart-M3 [71] targeted towards subscribing to resources in smart spaces (SPARQL and Wilbur Query Language (WQL) [80]), a Rete-based implementation employing distributed hash tables [116] (queries based on “tuple templates”) and a cloud-based prototype system [102] (SPARQL).

2.7 Stream reasoning

Reasoning in this context is defined as the capability to generate new knowledge out of facts and rules [9, 15, 19, 20, 46, 90, 105, 106, 121, 131]. Stream processing as a performance-enhancing technique outside the context of time-variant streams, but supporting reasoning on a subset of OWL is discussed in [44]. Stream reasoning has been used as a term to cover “agile, lightweight reasoning on rapidly changing information” [9, 15, 20, 46, 60, 70, 84, 90, 130, 131]. Temporal reasoning is further elaborated in [45].
In [63] reasoning frameworks are denoted *entailment regimes*. At the time of preparing Publication V support for RDF Schema (RDFS) [142] entailment was claimed for SparkWave\textsuperscript{34} [78] and Event TrAnsaction Logic Inference System (ETALIS) [56]. C-SPARQL website\textsuperscript{35} declared support for “simple RDF entailment”.

Reasoning over RDF graphs using forward-chaining rules and the Rete-algorithm is discussed in [80].

### 2.8 Benchmarking of stream processing platforms

Benchmarks for static RDF data include, e.g., the *Lehigh University Benchmark (LUBM)* [66], which tests reasoning, and the *Berlin SPARQL BenchMark (BSBM)* [27], which targets comparisons between RDF stores and SPARQL-to-SQL rewriters by offering a data generator producing both RDF and purely relational output. BSBM sets no requirements for reasoning and only includes queries, which have a straightforward conversion to SQL. For DSMS without RDF, *Linear Road* [12] is set as a variable tolling system of a fictional urban area. The queries address historical reporting, event processing in the form of accident detection and toll processing, which is dependent on the current traffic situation, including accidents. Linear Road carries many similarities to the *flower shop delivery* scenario presented in [53], for which many commercial (non-RDF) event processing platforms have presented implementations. For RDF stream processors, known benchmarks based on the data stream processing paradigm are, e.g., *LSBench* [82], *SRBench* [144], *CSRBench* [47] and *CityBench* [7]. Although all of the tasks in the benchmarks are not based on time windowing, the RDF stream processing engines, which only operate on time windows, are tested with the strategy of applying windows of specified lengths. Some challenges of benchmarking stream processing systems are discussed in [111].

#### 2.8.1 Published performance results

Due to different test cases (both data and queries) and hardware the performance results in other referenced publications are not directly comparable with the results in the included publications, but they serve as

---

\textsuperscript{34}With custom omissions and additions as declared in the referenced article.

\textsuperscript{35}http://streamreasoning.org/resources/c-sparql [Accessed Aug 11th 2016]
examples of performance demonstrated on other platforms. Testing on an AMD64 2x E5606 Intel Quad-Core Xeon 2.13GHz with 16GB RAM, [82] shows “comparable maximum execution throughput” values for the three platforms under study as Table 3. C-SPARQL achieves 1.63 – 10 triples/s, JTALIS\textsuperscript{36} 87 – 3,857 triples/s and CQELS 1,304 – 24,144 triples/s. ETALIS performance is tested in [10] with environmental sensor data, running on an SWI Prolog engine over Intel Core Quad CPU Q9400 2.66GHz, 8GB of RAM and Windows Vista x64. Excluding network delays and gaps in sensor input, ETALIS throughputs from 3,900 to 37,437 events/s are measured.

In [81, 104] the execution time is reported, but the precise number of triples in the dataset is not given. In [7] the latency figures are plotted, but throughput calculations are not included. Other referenced documents on RDF stream processor benchmarking [144, 47] do not contain performance results.

As observed also in Publication IV, the licensing terms of commercial CEP platforms typically do not allow the publication of performance results. An exception to this rule is Esper, whose permissible license was an important factor in selecting it as the comparison platform also for the study in Publication IV. Mendes \textit{et al.} [93] have published performance results of three commercial systems, one of them being Esper, but due to licensing terms the other two could not be named and the three systems were referred to as X, Y and Z. The reader does not learn, which one of the three is Esper and therefore not even the relative performance of Esper compared to the two other tested systems is revealed. The tests are executed on a server with two Intel Xeon E5420 (12M cache, 2.50 GHz, 1333 MHz FSB) Quad-Core processors, 16 GB of RAM and 4 SATA-300 disks running Windows 2008. In filtering tests throughputs from \(~200,000\) to \(~1,200,000\) events/s for row selection, and \(~10,000\) to \(~550,000\) events/s for column projection were measured. Aggregation over windows was highly dependent on window size, but reached up to \(~150,000\) events/s for a sliding window and \(~550,000\) events/s for a tumbling window for the fastest engine, while others showed performance of roughly half for the sliding window tests and about \(1/5 \sim 1/3\) for the tumbling window. Pattern matching results varied between \(~5,000\) and \(~50,000\) events/s. Performance in join tests varied strongly for difference window sizes,

\textsuperscript{36}A Java library version of ETALIS [65]

from \(\sim160,000\) events/s down to almost zero for window sizes larger than \(50,000\) events.

ETALIS performance has also been compared with Esper in 2011 [9]. In a “time sliding window” scenario ETALIS throughput is measured at \(\sim8,200 - \sim9,400\) events/s and Esper \(\sim7,000 - \sim8,900\) events/s, and in a “count window size” test \(\sim25,000\) events/s for ETALIS and \(\sim11,000\) events/s for Esper.

Between the non-RDF streaming platforms listed in Section 2.6.2, in [36] comparing Storm, Flink and Spark Streaming using Kafka messaging and JSON data the lowest latencies were demonstrated by Storm and Flink, while Spark was able to handle higher throughput. The tested throughput rates varied from 50,000 to 170,000 events/s. Lu et al. [87] present “StreamBench” and apply Kafka, Spark Streaming and Storm, but report their results in MB/s (not events, records or tuples per second), measuring throughput rates between 2 – 90 MB/s, Spark demonstrating about five times the throughput of Storm. In [59] Flink was found to perform faster than Spark or Hadoop, but also crash in certain tests. For Twitter, streams in the scale of “tens of billions of events per hour” are quoted [58]. White et al. [143] test ten parallel connections with up to one million events/s injection rate with a selective application producing output rates of up to 38,890 events/s.
3. Events, states and background knowledge

“Temporality is obviously an organised structure, and these three so-called elements of time: past, present, future, must not be envisaged as a collection of ‘data’ to be added together... but as the structured moments of an original synthesis. Otherwise we shall immediately meet with this paradox: the past is no longer, the future is not yet, as for the instantaneous present, everyone knows that it is not at all: it is the limit of infinite division, like the dimensionless point.”

– Jean-Paul Sartre, Being and Nothingness, 1943

To properly act on events, information on instantaneous events often needs to be complemented with other types of information, such as the state of the system prior to or after an event occurs, or some more static background knowledge, which may define rules or expectations for the interpretation of the event. This chapter contains definitions for the different types of information utilised in this study, classified by the time of when such facts are (asserted to be) valid.

3.1 Event objects

An event object $E_T$ is defined here as a set of one or more attributes asserted to describe a snapshot of a system of interest specified as event type $T$. An event object may incorporate zero (Publication V), one (PI and PIII) or more (PIV) timestamps, which can be selected for processing (e.g., filtering, time windowing) as required. The creation of event objects from measurements and other data sources is illustrated in Fig. 1 of Publication II.

$^1$In [89] an event type is defined as “a class of event objects.”
Relations between the different categories are illustrated in Figure 3.1. *Event category* is a general classification for all event objects related to their origin and relationship to other event objects, whereas *event type* represents a context-specific grouping related to the purpose and attribute structure of an event object. It should especially be noted that the category definitions do not automatically imply anything about event object encoding. The category of an individual event object can be made explicit, if selection between categories is required by the application.

A simple event may contain a lot of attributes (from the same source) and it may therefore appear “complex”. The creation of a complex event in itself may summarise a complex chain of events, but the resulting event object may not require many explicit attributes and it may therefore appear “simple”.

Figure 3.2 shows an example, where the event objects describing the presence of individual meeting participants and the current time are compared to background information on meeting starting time and intended participants to derive that the meeting started as scheduled.

Publication II defines the structural elements to separate event object contents to two distinct components:
• **Header:** the necessary information for processing the event object in a format, which is understood by the current EPN. As examples, Figure 4.1 shows a label, the event object time and references to both an event object *component* (as in a composite event) and an event object *constituent* (the event object being summarised in a complex event).

• **Body:** payload transported together with the event object, but not processed and not necessarily understood by the current EPN.

Separation of body payload from the header can be justified for systems involving multiple EPN, where all of the content is not expected to be semantically understood and used by every EPN. Such cases arise, e.g., in the design of protocol stacks like the Open Systems Interconnection (OSI) Reference Model [43], where the lower protocol layer is expected to transfer higher layer data as payload without any access to or manipulation of the content. For the specific case of RDF, Publication II lists the example of the payload containing vocabularies, which are not known in the current EPN. The event processing systems implemented in all other included publications (I, III, IV and V) involve only a single EPN and do not separate header and body components.

Different event sources may incorporate different elements to event objects, even when the primary purpose (such as the measurement of temperature) and therefore the application-dependent type is the same. Therefore event objects in this study are expected to be *heterogeneous* – possibly containing optional elements – as illustrated, e.g., in Fig. 1 of Publication III.

### 3.1.1 Time in events

The real-time processing of event objects is often closely connected to the perceived *time of occurrence* of the real-world events: time gives an understanding of the order of events, their simultaneity within some context-dependent level of accuracy or their age at the time of processing. Unfortunately the timing of event object creation in a distributed environment may not always be fully clear:

- Some sensors may not have a clock input
- The clock circuits of all asynchronous distributed systems may drift.
The issue can be mitigated by network synchronisation protocols. [38, 79]

- Power outages may impact clocks

In addition to the original sensor source, any other steps of processing an event object may have a time reference, which can be associated with the event. In Publication II the following attributes are defined:

- **Time of sampling** based on the clock of the sensor. May be incomplete due to the reasons listed above.

- **Time of entry** to a data stream. May not correctly reflect the order of sampling due to variation in transmission delay from the sensor platform to the data stream.

- **Time of arrival** in an event processing system. May not correctly reflect the order of entry to data stream or the order of sampling due to transmission delay variation.

Different event processing tasks have different requirements. In some cases the best available estimate of the time of sampling is important, even if there is no value available for every sample. In other cases it is imperative that every event object has a timestamp, even if that timestamp may not accurately reflect the order of event object creation. Some event processing agents need to distinguish between time of entry to the data stream, while others depend on the time of arrival in the event processing system. Therefore platforms for event processing should support the association of multiple timestamps with individual event objects to allow the platform user to select the best time reference for a given task.

For example, the event objects in Publication IV have two timestamps:

- `eventRecordedAt` from the clock of the Radio-Frequency IDentification (RFID) event reader.

- `eventOccurredAt` reflecting the time of entry to the data stream.

For the task in Publication IV `eventOccurredAt` is used, because the in-
accuracies in event reader clocks dominate over any transmission delays, and the order of event objects joining the data stream is important.

To assist in creating event processing applications, which can properly utilise the time information associated with events, either a formal or informal description (ref. Section 6.5) of the event stream should be made available.

It can be argued that with sufficient accuracy any natural event will take place during an interval. Why would event objects only be assigned single timestamps by each point of reference? An event object – sensed or synthetically generated – is always the result of a process in a discretely operating computer. For any measurement or other execution of a computer program there is always a singular point in time in the framework of that platform, with accuracy suitable for the task, when the event object entity becomes complete. For the examples in this dissertation the time of trigger of the sensing of a phenomenon – not the duration of the event – is used as the timestamp.

### 3.2 States

Both in computer programming and other contexts a state is used to describe the condition that something (or someone) is in at a specified time. To keep the definition and format of event objects uncluttered, the term state is hereby adopted to describe any condition, which has a beginning and an end, either one of which may stretch to infinity.

State entries and exits are typically triggered by event objects or combinations thereof. As every event object $E_T$ describes an implicit state of a system, a continuous process may pass through an infinite number of implicit states during its lifetime. It depends on the task, whether a state $S_x$, where $x$ uniquely identifies the state in the system of interest, should be made explicit. When made explicit, $S_x$ is boolean-valued: the system either is in state $S_x$ or it is not.

Unless specifically defined as complements, states are not exclusive: multiple states can be true simultaneously. The start and end times of a state always exist implicitly, but it is a task-dependent design choice, whether they should be explicitly recorded or saved as $S_x(τ_{start}, τ_{end})$. It is also a design choice, whether a state should be given a name (the “nearby” state in Publication I) or simply an index (current progress of the event pattern of EPAδ in Publication III).
States are used to label and track continuous processes, which could also be referred to as “continuous events”. The entry and exit conditions of a state can be based on any combination of the reception of events, event patterns, attributes (including time) or results of computer processing based on attributes. The true-condition of a state does not limit any other changes in the system: any number of (related or unrelated) input event objects may be independently processed before the exit-condition of a state is met.

If event objects were required to incorporate durations, the timeliness of event streams would be severely impacted: an event could only be added to a stream once it is completed and the end time is known. Creating a log of state entries and exits is a good way to document history, but as an object of streaming it would contradict requirements for real-time operation\(^2\). In the trials conducted within this body of work event objects are streamed and states exist only within the programs processing the streams.

On the borderline of the split between events and states Publication II, p. 6 incorporates also a parameter for the time of event object validity. This is interpreted as the anticipated validity known at the time of event object creation, and is therefore just one of many parameters which can be associated with an event object.

### 3.3 Background knowledge

In addition to events and states there is information, which contains no associated time of validity. Such statements are currently asserted to be true, but it may not be known when they were generated or when they will expire. The term *background knowledge* is used in this study to describe such information.

Typical examples on Description Logics (DL) do not consider the time validity aspect. If nothing is stated about the validity time of an assertion, both TBox and ABox assertions would have no associated time of validity and belong to background knowledge. Both TBox and ABox assertions could also change in time, in which case their assertions could be modified by events, and their current values would define states, either implicitly or explicitly, depending on design choices of the application. The more

\(^2\)Indicatively search term *event stream* retrieves seven times more results than *state stream* in Google (Retrieved Mar 4th 2016)
typical case would be to have infinitely valid TBox assertions and time-varying ABox assertions.

Publication V is slightly misusing blocks of ABox data in LUBM, consisting of university departments, in treating them as time-varying data. This approach was necessary to compare the performance between an event stream processor (INSTANS) and non-event-based RDF platforms with reasoning. No LUBM query requires matching between departments. Therefore processing departments one by one does not impact the results, which were confirmed identical, even though LUBM does not originally specify any time of validity for the department-related assertions.

3.4 Example of events, states and background knowledge

An example of the relationship between events, states and background knowledge, based on the test case of Publication I, is illustrated in Figure 3.3. The application notifies each application user when a friend is detected nearby. A notification is used here to describe an event object, which triggers information to be shown to a user. The point of reference for notifications is the application user interface.

First, we have the background knowledge that Jim is a friend of Alan and that friendship is considered symmetrical, implying that Alan is also a friend of Jim. As location update event objects from Jim and Alan are received, it is detected after location1 from Alan that Jim and Alan are in close proximity. As that state was not active before, the platform generates event objects, which trigger notifications of the nearby friend to be shown to Jim and Alan and establishes a Jim closeTo Alan state. While the next location update events from Jim and Alan remain close to each other, no new notifications are triggered as the Jim closeTo Alan state is already active. Location3 from Jim eventually indicates that Jim is moving away from Alan's latest known location and the Jim closeTo Alan state is terminated. If required by the task, the duration of the state could now be recorded and / or streamed. Next detection of Jim and Alan being in close proximity would trigger new “nearby”-notifications and a re-establishment of the state.
Figure 3.3. Relationships of events, states and background knowledge based on the example of Publication I.
4. **RDF-encoding of structured objects**

“No object is so beautiful that, under certain conditions, it will not look ugly.”

– Oscar Wilde, *Lecture to Art Students*

All event objects in the embedded publications have been encoded using RDF. Examples of RDF events encoded in Turtle in the embedded publications can be found in Publication I, p. 4 and Publication II, p. 8. Representations of comprehensive, heterogeneous events require the use of a flexible number of triples. Therefore event objects $E_T$ are in this work represented by graphs $G_T$.

Figure 4.1 shows a simplified illustration of the flood warning example event graph from Publication II, p. 8. Colons (“:”) separate prefixes, which are shortcuts of Internationalised Resource Identifier (IRI) addresses to ontologies giving global definitions to the namespaces being used. Prefix “ep” refers to the *event processing ontology design pattern*¹ (ODP) defined in Publication II, the base prefix “:” refers to our namespace in the instans.org domain and the other prefixes “rdf”, “rdfs” and “xsd” belong to the foundation of the Semantic Web. The black :floodWarning0001 is the *root node* of the event, and the only IRI subject in Figure 4.1. Blank nodes “_:0” and “_:1” are used here to connect the fields of the event object header and body to the original subject.

### 4.1 Event Processing ODP

RDF encoding of composite events requires a solution, where component event objects can be encapsulated as integral parts of a composite event object but still remain complete so that they can later be restored to in-

dependent objects by an EPA. If the composite event object header would simply contain a reference to the subject of an event object component, the component would remain intact but a new observer joining the stream would not be able to detect that a received event object is an encapsulated component of an unknown composite event. If the root node subject of the event object component would instead be substituted by a blank node referenced in the composite event object header as in Figure 4.2 a), a late-joining observer would be able to detect that the received event object belongs to an unknown entity. The original subject of the component would, however, be replaced by the blank node and lost, and the component could no longer be fully restored as the original independent event object.

The event processing ODP presented in Publication II can be used as a modular component to extend the definitions of Event-F [112] and SSN [85]. It offers tools for bi-directional linking: complex and composite event objects can link to their child event objects, and the children can be linked to their parents. Figure 4.2 b) illustrates a bi-directional reference, where both the composite and the component event object reference each other, but the component remains otherwise unchanged. When composite events are constructed accordingly, a late-joining observer can detect if the first-received event objects are components of an unknown composite event object. The approach in Figure 4.2 b) is the recommended approach to encode composite event objects in RDF.

The following contributions to the representation of events in RDF are
made specifically by Publication II:

- Separation of events and event objects.

- Support for event object payload through predicates `hasEventObject-Header` and `hasEventObjectBody`.

- Support for complex events through predicates `refersToEventObject-Constituent` (header), `hasDirectSubEventObject` (no-header, non-transitive subproperty) and `hasSubEventObject` (no-header, transitive superproperty) from the side of the higher-level complex event object and through predicates `isDirectSubEventObjectOf` and `isSubEventObjectOf` from the side of the lower-level event object.

- Support for composite events through predicates `refersToEventObjectComponent` (header), `hasEventObjectComponent` (no-header, non-transitive subproperty) and `hasEventObjectPart` (no-header, transitive property representing partonomy) from the side of the higher-level component event object and through predicates `isEventObject-ComponentOf` and `isEventObjectPartOf` from the side of the lower-level event object.

- Support for multiple timestamps of an event object with predicates `hasEventObjectSamplingTime`, `hasEventObjectApplication-
tionTime (time of entering a data stream), hasEventObjectSystem-Time (arrival in the event processing system) and hasEventObject-ExpirationTime (a priori information on the validity of the object).
5. Streaming RDF

“If it weren’t for the rocks in its bed, the stream would have no song.”

– Carl Perkins, Interview in “Rolling Stone”, December 1968

To process delay-sensitive events, intermediate storage and buffering have to be minimised and the event objects delivered in one or more queues. An event stream is defined here as a queue of discrete event objects $E$ of type $T$:

\[
\begin{align*}
&\ldots \\
&E_{T_i}(i) \\
&E_{T_{i+1}}(i + 1) \\
&\ldots \\
&E_{T_{i+x}}(i + x) \\
&\ldots 
\end{align*}
\] (5.1)

Index $i$ is a positive integer monotonically increasing index for subsequent event objects used in the following definitions. If all event objects in a stream are of the same type:

\[\forall i, j | T_i = T_j\] (5.2)

The stream can be characterised as homogeneous (e.g., the streams of location updates of Publication I and Publication III). If the stream contains events of multiple types:

\[\exists i, j | T_i \neq T_j\] (5.3)

The stream is heterogeneous (e.g., the stream of commissioning, packing and shipping events in Publication IV). We therefore observe that heterogeneity in a stream may exist on two levels: in the inclusion of optional attributes of an event type to individual event objects and between the event types present in a stream. Whether an individual event object
$E_T(i)$ describes an actual change in the system, depends on whether any attribute values have changed since the previous snapshot of the same type $E_T(i - x)$.

### 5.1 Solutions for timestamps

Timestamping of individual triples (as in TRDF, Section 2.5.1) carries the assumption that an event consists of a single triple. This assumption is very limiting, as it means that an event can only express one value for one attribute. One way to construct more elaborate events is to define that two or more RDF triples with the same timestamp belong to the same event. This is not a working solution:

1. Two or more triples from different sources may carry the same timestamp and be mistakenly understood as belonging to the same event.

2. The completeness of an event can only be determined after the next triple with a later timestamp arrives.

3. Any copy or move operations on the stream may unintentionally merge or split events.

Use of external timestamps – for either triples or graphs – introduces the following problems:

- No syntax compatibility with existing RDF tools\(^1\). Annotation support through quads has not gained popularity in RDF tools. When quads are supported, the fourth element normally refers to a named graph ($<s, p, o, g>$) [141].

- Specifying the exact number of timestamps (e.g., “1”) as a part of the serialisation format reduces flexibility.

Instead of defining new formats for external timestamps, the approach in all of the embedded publications has been to incorporate timestamps – when such exist – into the events themselves. The benefits are the

\(^1\)Not a valid argument, if using reification [142]. However, none of the referenced RDF stream processing tools support timestamps through reification.
reverse of the above: The resulting data streams are fully syntactically compliant with all existing RDF tools, and any number of timestamps per event object can be incorporated as required by the task without breach of compatibility. Limited processing of the streams such as simple store-and-forward can be carried out without knowledge of the timestamp semantics. The required accuracy of timestamps also varies from application to application, typically between nanoseconds and days.

For time-aware processing of the data stream the chosen solution has to be shared by all participating processing agents. Otherwise the link from the timestamps to the rest of the objects is lost. If time is not part of the inferential semantics of the representation (as in, e.g., TRDF), the external semantics of the timestamps have to be communicated to all participating nodes (ref. Section 6.5).

The benefits of external vs. embedded timestamps are also coupled with stream processing platform implementation, which will be discussed in more detail in Section 6. INSTANS, which is used in the trials in this study, does not provide language extensions for processing stream windows. As any references to timestamps in the queries are coded manually in SPARQL, any timestamp appearing in an event object data can be utilised. Most other RDF stream processing platforms (e.g., [9, 17, 30, 75, 78, 81]) provide language extensions for the processing of time windows. Corresponding platform implementations either assume a fixed format for the timestamp in the stream or use the system time of the host computer to assign timestamps based on triple arrival time. Some implementations separate the steps of stream-to-relation [11] and query processing onto different software modules or platforms, which makes it even more difficult to parse an RDF-format event object only for the purpose of decoding a timestamp at a point in the processing chain, where otherwise no access to the RDF-format data would be needed. On such a system it is clearly more straightforward to use a stream format with external timestamps. INSTANS carries out continuous query processing in one module, enabling the matching of timestamps from event data.

The superiority of different solutions can be debated also from a philosophical standpoint. If timestamps are statements about the validity of the event object data, they can be argued to belong to another meta-level than the event object content. A counter-argument is that, e.g., the location of the measurement for an event object is also data – spatial metadata – about the event, but it is still typically encoded internally within
the event object. Timestamps have reached a special status due to history in data stream processing, but in some tasks the first step of filtering a stream can be based on another parameter, e.g., spatial location.

The experiments carried out in this body of work have not revealed any substantial reason, why timestamps should receive special treatment compared to any other event object attributes.

5.2 Heterogeneous events in an RDF stream

The completeness of an event object must be explicitly recognised by the processing platform. When operating with heterogeneous events, which require optionality in queries depending on whether a certain property is present in an event object or not, care must be taken. The issue can be observed in the extended SPARQL event processing network in Publication III. Unless events are properly handled as blocks, multiple unintentionally different results can be produced by a single query from a single event object, if the underlying rule processor executes between each addition of an optional element.

5.2.1 Issues with streaming Turtle

In the first two publications of this dissertation event streams were sent as continuities of the Turtle format [22] as shown in Publication I, p. 4 and Publication II, p. 8. While this remains a working solution for limited scenarios, more complex scenarios brought up issues with punctuation.

As addressed in Publication III, p. 4: rule #6 of the Turtle grammar [22] defines “triples” as a group of triples continuing until open forward references to blank nodes have been exhausted. This solution worked in the context of Publication I & Publication II, but was found to introduce a dependency on the order of triples in an event object.

The issue is exemplified in Figure 5.1 showing the the same event object from Publication III on both left and right. The order of triples on the left would separate the location coordinates into another group of triples, as they do not contain any open forward references and therefore block 1 appears complete to the processing platform. On the right the first block has an open blank node forward reference to the location coordinates, enabling the event object to be parsed as one block of triples, as intended. Proper handling of heterogeneous event object graphs clearly requires iso-
lation either through punctuation with externally defined semantics or a level of graph encapsulation.

5.2.2 Streaming a dataset of graphs

From Publication III onwards the issue indicated in Section 5.2.1 was resolved using the *TriG RDF dataset language* [26], which became a World Wide Web Consortium (W3C) recommendation during the time of the study. TriG defines communication of RDF datasets by *graph statements*, which are pairs of an IRI or a blank node and a graph surrounded by `{ }`. The IRI and blank node prefixes can also be omitted, in which case the graph statement corresponds to the default graph of a dataset. The difference between streams of Turtle and streams of TriG is not one of syntax, as Turtle is a serialisation format for RDF and TriG is a serialisation format for RDF graphs serialised as Turtle.

Encapsulating each event object as a graph in a TriG dataset removes all ambiguity regarding punctuation. As TriG support was added to the INSTANS platform, it was done so that each graph represents a block, which can be configured for joint processing. When a client consumer joins a stream during transmission, it is possible to unambiguously skip triples until the beginning of the next graph, which will be the next complete event object in the stream.

Subsequent studies of event streams were carried out using the TriG format. Examples of TriG-encoded event objects can be found in Publication III, p. 4 and Publication IV, p. 3.
6. Event stream processing

“Whoever fights monsters should see to it that in the process he does not become a monster.”

– Friedrich Nietzsche, Beyond Good and Evil, 1886

INSTANS, developed as two separate and independent versions by Abdullah [3] and Nuutila at the Department of Computer Science¹ of Aalto University, is an RDF stream processor with support for asynchronous query networks, which can be used to implement EPNs. The first version, coded in the Scala language² [98], was used in Publication I, while the latter version coded in Common Lisp³ [120] was used in all the other embedded publications.

Stream processing in INSTANS is truly continuous: each incoming triple or event – depending on the configuration – is processed to an intermediate result in the Rete-network and query results are output as soon as all the required inputs are available. An example of the Rete-network formed in INSTANS is shown in Publication I, p. 7 (Figure 1). Despite the use of the same term – continuous – INSTANS operation is fundamentally different from the operation of DSMS systems, which repeatedly execute persistent queries for each new window. Queries in DSMS are processed repeatedly but not continuously.

Continuous operation also sets challenges. In Section 5.2.1 the handling of incoming optional data elements was discussed. The same is true for the removal of an event. With INSTANS the platform user has no direct control over the order, in which the Rete-network executes simultaneously activated rules. If the network is executed after every operation, deletion of events from the input queue may trigger unwanted and faulty query

²http://scala-lang.org/ [Accessed Sep 3rd 2016]
³https://common-lisp.net/ [Accessed Sep 3rd 2016]
answers, when optional elements are removed. INSTANS addresses these challenges by a user-specified execution policy, configured using an ordered list of operations to be carried out with every addition of an input unit. The input unit can also be specified as a triple, a block (ref. Section 5.2) or a file.

### 6.1 Mapping of EPN elements

In the CEP vs. DSMS axis INSTANS positions itself on the CEP side as it does not force stream windowing and supports event patterns and hierarchical rule networks of arbitrary sizes, tested in Publication V with up to 103 simultaneously active rules. The composite and complex event structures shown in Publication II are also supported in INSTANS. The correspondence of EPN elements between CEP [53] and the SPARQL conventions used in the included publications are:

- **An event channel** is represented by a named graph.

- **An event producer** is an input event stream (of RDF graphs) to the current system.

- **An event consumer** is an output stream, typically a file or console output.

- **A global state element** is reference data which can be accessed either as information stored locally in a graph or through a federated query (Publication III Figure 7).

- **An EPA** consists of (a network of) SPARQL queries and rules carrying out a specific task. It receives input from event channels, manages internal state in RDF graphs by means of SPARQL update operations (INSERT, DELETE, etc.), accesses global state elements by means of (optionally federated) SPARQL query and update operations, and outputs results to other EPAs using event channels by means of INSERT operations and to event consumers using SELECT and CONSTRUCT queries.

- **An EPN** can have multiple event channels, event producers, event consumers and EPAs.
6.2 Defining event patterns

The tests carried out for the included publications typically involve processing of a pattern of two or more events:

- Publication I: Proximity of two friends is detected by comparing location updates of individual persons with background knowledge of friendship relations.

- Publication III: The pattern detect module searches for a pattern of movement from consecutive location updates.

- Publication IV: Manufacturer logistics monitoring is searching for stolen products or counterfeits using patterns of scan results between two scanning locations.

As INSTANS uses specification-compliant SPARQL without proprietary extensions, there is no specified pattern language [53]. Consecutively, any pattern that can be expressed using a network of SPARQL queries\(^4\) can be recognised. This includes, e.g., the following types of patterns (partially aligned with the list in [53]):

- **Conjunction**: All events of a pattern are detected.

- **Disjunction**: Any event of a pattern is detected.

- **Negation**: An event or pattern is not detected (before a defined timeout).

- **Set**: A pre-defined set of events is detected (within defined time limits).

- **Sequence**: A pre-defined sequence of events (some of which may be negations) is detected (within defined time limits).

- **Threshold**: An aggregation operation performed against the set of participating events. Any mathematical operations available in SPARQL can be used. Aggregate values (min, max, count, average) can be computed (as demonstrated in Publication III). Multiple attributes from the

\(^4\) Assisted by graph storage as needed.
events can be separately compared.

- **Trend**: Patterns following defined trends (e.g., increasing or decreasing) can be detected.

As a network of SPARQL queries is closer to a programming language than a pattern specification language, the limitations of expressive power are not very strict. The absence of square root in SPARQL (ref. Publication III section 4.1) means that spatial or spatiotemporal patterns involving distances typically require an extension function to calculate the distance. Detection of complex patterns is possible, but not always practical. Especially dynamic handling of contexts, e.g., for tracking the progress of multiple overlapping pattern sequences requires more effort with managing dynamic references to graph storage in SPARQL than in typical modern programming languages.

### 6.3 Event processing with networks of SPARQL

The design choice of INSTANS to be based on standard-compliant SPARQL implies that most of the explicit stream processing or event processing query constructs present on other platforms require a specific implementation, often by means of a network of SPARQL Update rules. Comparing with the set of query language features listed in the currently available draft requirements and design principles of the W3C RSP Community Group (ref. Section 2.5.1), and the Esper 5.5.0 reference manual [52], the support of event processing features on INSTANS is described as:

- **Input stream selection**: One input, multiple streams supported as separate graphs (PIII).

- **Stream join or split**: Supported by SPARQL Update for named graphs (PIII EPA5 and EPA7).

- **Event schema definitions**: Neither required nor supported.

- **Selection and comparison of event properties**: As supported by SPARQL.

- **Subqueries**: Either as rule networks or through the SPARQL subquery
Event stream processing

mechanism [137].

- **Dynamic event properties**: Supported through SPARQL OPTIONAL-clause.

- **Time input**: Rule network with graph memory (PIV Section 4.2).

- **Stream windowing and aggregate processing**: Rule network with graph memory (PIII EPA6).

- **Context partitions**: Rule network with graph memory.

- **Patterns**: Rule network with graph memory as supported by SPARQL (PIII EPA8).

- **Background information access**: Provided as input to INSTANS (PI, PIII and PV) or through SPARQL federated queries (PIII EPA3).

The differences between INSTANS (a stream processing platform based on Semantic Web technologies) and Esper (not based on the Semantic Web) are discussed in Publication IV. As can be seen in PIV Figures 2 and 3, the basic matching of event attributes to query templates is similar. INSTANS supports OWL ontologies and RDF data, Esper supports XML namespaces and XML data (among other formats). The main differences between the two platforms regarding the query language are:

1. Esper EPL supports a rich set of extensions for event stream processing. INSTANS relies on query networks.

2. Esper platform and EPL support a system-level clock with external input. INSTANS relies on a specific implementation using query networks.

Nothing prevents the addition of EPL-like event processing extensions and system-level clock support to INSTANS. Likewise, support for RDF data could be added to Esper. The main difference between the two platforms, a constantly evaluated rule network (INSTANS) vs. an event-triggered processing chain (Esper, Publication IV) is an implementation choice not dictated by the format used for event data or queries. There-
fore, as discussed in Publication IV Section 6, the main differences between event processing approaches with or without Semantic Web technology support originate from other properties of Semantic Web technologies, especially the support for reasoning (PV) and federated queries (PIII).

### 6.3.1 Advances in event processing

Several methods of event processing in the SPARQL domain appear for the first time in the included publications. Publication I presents the method of asynchronously, continuously and hierarchically processing RDF-encoded multi-triple events using query networks constructed out of combinations of specification-compliant SPARQL queries and update rules. Publication III builds upon the approach of Publication I by presenting examples of all the eight different types of EPAs appearing in [53] (Publication III, Figure 4): stateless and stateful filters, enrich\(^5\), project, split, aggregate, compose and pattern detect.

Publication II, p. 9 presents a method (by the third author) for finding composite event components by querying with property path expressions [137] through arbitrary levels of known pairs of predicates using the asterisk operator as in (\(p1:p2\))^\(*\). This approach enables matching of the root triples of the component events of a composite event. Publication II, p. 10 also presents a method (by the first author) for matching the contents of the component events by using nested \texttt{OPTIONAL} clauses. While this is a working solution, it is concluded also in the paper that it has the limitation of requiring \texttt{a priori} knowledge of the maximum number of supported nesting levels, as the nested clauses need to be explicitly written into the query. In later work this limitation has been overcome by the use of the TriG dataset format and a separate graph for each event, as explained in Section 5.2.2. The dataset-approach allows to match an event as the complete content of a graph – “\(\text{\textit{s p o}}\)” – once the appropriate graph is found. Even though the scenario of Publication II has not been reproduced in later trials, this type of matching an entire event can be observed in Figures 5, 6, 7 and 8 of Publication III. Combined with the property path expression of Publication II this approach solves the processing of composite events of arbitrary nesting levels.

As explained in Section 5.2.2, the TriG-format is used in Publication III

\(^5\)using a federated \texttt{SERVICE}-query over the internet
and Publication IV to encapsulate each event as a named graph. Publication III introduces a method, where the graph name is used both for identifying event channels and individual events by filtering the graph names accordingly. The TriG format [26], however, requires neither distinct names nor any names for the incoming graphs. Therefore TriG can also be used only for punctuation. If graph names are omitted entirely, all graphs are interpreted as belonging to the main graph. If all incoming graphs have the same name, they can all be interpreted as belonging to the same named graph (= event channel). Therefore the partial filtering and construction of unique names for all events in all channels used in Publication III is not mandatory. Still, it can be seen as improving event traceability for debugging.

6.3.2 Challenges of SPARQL in event processing

SPARQL Query language [137] is originally intended to process queries over finite RDF datasets available, e.g., as Linked Open Data, which is a distributed network of information exposed in the World-Wide Web. Even though SPARQL generally fits well to a streaming context, it also has some characteristics, which require special interpretation. This concerns especially solution sequence modifiers and aggregates [137]. While this dissertation does not contain a systematic analysis of the semantic interpretation differences to SPARQL semantics forced by stream processing, examples appearing in the included publications are listed below for reference:

- **ORDER BY** is inherited from SQL for ordering the answer set. In continuous stream processing solutions are expected to be individually output immediately when they become available. The solutions either need to be output one-by-one (in which case the ORDER BY solution modifier has no impact) or older solutions need to be stored and repeated in a newly sorted order every time a new solution becomes available. Maintaining the entire set of solutions indefinitely may eventually cause memory issues, depending on the absolute quantity of solutions required by the task.

- **LIMIT** limits the number of solutions output by a query. The logical interpretation would be that solutions are output up to the specified limit, after which no more solutions will be made available before a restart of
Event stream processing

- Aggregates calculate aggregate values (sum, average, maximum, minimum, count) over the answer set. The logical interpretation also applied in INSTANS for the built-in aggregate operators in SPARQL is to output a new value every time the aggregate is computed based on new input.

In Publication III it was identified that even though SPARQL 1.1 supports TriG input, there is no specified way to create dataset output apart from writing to a graph using an INSERT-clause from SPARQL Update, which in the case of INSTANS is a memory-internal operation. Using the same format as INSERT from SPARQL Update, dataset output capability was added to CONSTRUCT queries for INSTANS. No side-effects on the syntax were detected. Consequently no problem has been found for adding this capability to future versions of SPARQL Query specifications.

Publication III modelled event channels between different EPAs as named graphs (ref. Section 6.1), identifying also each event uniquely by embedding a number to the graph name. When each event forms a separate “channel”, incoming events and channels need to be processed using suitable filtering. In Publication III \(<Eve1>\) becomes \(<Poststateless-Eve1>,<Poststateful-Eve1>,<Translated-Eve1>\) and \(<Projected-Eve1>\). SPARQL does not directly support wild card matching for GRAPH-statements. Instead separate FILTER-statements are applied to match the beginning of each graph name to match events in the appropriate channel. For the outgoing events, appropriate names are constructed and bound to variables, which are used as graph names for the outgoing INSERT-statements. Examples of this method are found throughout Publication III in Figures 5, 6, 7, 8, 9 and 11. Wild card filtering for incoming GRAPH names and possibility to directly use IRI() function output for outgoing graph names in INSERT operations would reduce extra steps and variables in commonly used queries for event processing.

SPARQL Update [138] was created as an additional component in version 1.1 release of the SPARQL specification set for the purpose of graph maintenance. The INSERT and DELETE operations, which attach to a query form to select the data to operate upon and together can carry out replace-operations are used extensively in all of the attached publications. INSERT-rules move, copy and filter data to named graphs carrying out the task of event channels in an EPN, as well as provide temporary value
storage for queries and query networks. DELETE-rules carry out cleanup operations both for input events and event channels, to keep memory consumption stable and prevent unintended combinatoric matches between old and new events. Also INSERT DATA, which inserts inline RDF -data to the specified graph from a SPARQL query file, is used in, e.g., Publication III and Publication V to initialise values.

Using SPARQL Update as a rule language for event processing has presented some challenges. Because SPARQL Update and Query originate from separate specifications, there is no jointly specified rule and query format. Consequently, while SPARQL Update rules can modify graph contents, a separate SPARQL Query is needed to produce output. The missing support for a rule which would – based on certain input conditions – both produce output and modify graph content by, e.g., deleting or modifying the input, results in the need to duplicate query graph patterns. The issue is discussed in Publication III, p. 9 and shown in PIII Figure 15, where the WHERE-clauses for the result output rule and reset index rule are identical. There is no practical reason why the DELETE and CONSTRUCT clauses could not be attached to the same WHERE-clause other than the current SPARQL grammar [137], which separates between a Query and an Update and does not allow mixing the two.

### 6.4 Challenges of DSMS

Event patterns do not match well with the DSMS time window paradigm. The problems are discussed in Publication I, p. 5 and Publication IV, p. 4 (and Figure 9):

- **Time windows force periodic computation** of queries even when there are no qualifying incoming events\(^\text{6}\).

- **Tumbling windows** (non-overlapping) may split event patterns, resulting in false or missing solutions.

- **Overlapping windows** produce both duplicate solutions and false positive results on window borders.

\(^6\)Some implementations trigger query processing based on input ("eager execution strategy" [82]) and are able to skip processing, if there is no input during a window interval.
• **Notification delay** in DSMS is typically dominated by the window length (Publication I) and therefore unstable.

• **Configuring shorter windows** in an attempt to improve notification delay performance results in more redundant computation and duplicate results.

Ways to improve DSMS support of event patterns can be envisaged:

• **Persistence of state** between windows allows to instantiate a window-independent state shift based on the reception of an event. Described for TEF-SPARQL [75] but with no existing reference implementation at the time of writing.

• **Auxiliary streams** can be used to transfer results output from one window as input to another window. Not commonly applied in the references, but demonstrated for comparison in [75].

• **Filtering the relative position of an event object in a window** could be used to confirm that event objects match a query only in those time windows where the rest of the expected event pattern would also fit within window boundaries. No support has been found in any of the referenced DSMS platforms.

Although methods to circumvent some limitations of the DSMS approach for event pattern processing exist, they increase complexity and thereby make the resulting applications more error-prone. Additionally, synchronous processing of windows quickly increases delay compared to asynchronous event-based processing for hierarchical approaches. For these reasons comparison to a DSMS system (C-SPARQL) was only carried out in Publication I.

DSMS and CEP are not mutually exclusive paradigms and neither are all the platforms, on which they run. Esper supports a special window type called **batch window**, which can be used for calculating aggregate values over time windows. Also on INSTANS aggregates can be calculated with rule networks, as demonstrated in Publication III, p. 7 (Figure 10 / 7The time from the reception of a triggering event object to the availability of the corresponding query answer.

60
**6.5 Stream descriptions**

The alternative approaches of either incorporating timestamps into events or attaching them to the format of the stream (Section 5) bring up the question of how an event processing platform recognises these timestamps. In the current situation:

1. Timestamps within the events are taken explicitly into account in the SPARQL rules, as shown, e.g., in Publication III Figures 5, 6 and 10. This approach requires tailored support in the queries (or a tailored pre-processing rule converting incoming timestamps to a common format).

2. External timestamps are hardcoded into input stream processing, e.g., for the purpose of extracting stream windows. This approach has limited flexibility towards the number and semantics of timestamps available in the stream.

One approach to improve upon item 1 would be to specify a format for *stream descriptions*. The root IRI advertised as the address of a stream would ideally point to a formally encoded stream description, which would contain enough detail on the elements of the stream to automate query generation as well as provide the actual stream web addresses of which there could be more than one, in case the stream is available in multiple formats. E.g., the optionality, accuracy, reliability, reference point of measurement and unique predicates of each timestamp could be specified.

The publications in the present dissertation do not address this topic, but an overview of suitable methods has been prepared as a B.Sc. thesis (in Finnish) [100] supervised by the author of this dissertation. The *evented API* also talks about an “event dictionary”, which “should list possible attributes along with the syntax and semantics.”

6.6 Memory management

An infinite stream of data can potentially fill any memory or database. One target for stream processing is to provide means to release resources which are no longer needed for future processing. The DSMS paradigm allows a straightforward approach: Once a given time window has been processed, the window data and all intermediate constructs can be discarded. This is simple and efficient, but at the same time it would be severely limiting towards the event processing tasks handled in Publication I, Publication III and Publication IV.

In event processing the timespan of completing an event pattern is task-specific and highly variable. The task may involve, e.g., seasonal changes, in which case it would be perfectly plausible that certain events persist in the system for more than a year. At the same time, if the lowest level of events are created at the rate of, e.g., 100 events per second and notification time requirement is set at one second, computing year-long time windows over the complete stream every second would not be efficient. It is easy to see that the storage time requirements for events need to be specific not only to the task but also to the event object type. The platforms tested in the embedded publications offer some methods of memory management, which have been used as follows:

- **Esper** provides a window mechanism, which can keep an incoming event accessible for a predefined time and emits notifications when events enter and exit windows. This mechanism is discussed in Publication IV, p. 4. Events not captured by any window can only be referenced upon entry to the system.

- The operational policy in **INSTANS** can be configured as a list of operations with the command line parameter `rdf-operations`. One of the operations is `remove`, which can typically be used to remove incoming events after all the dependencies have been computed into the Rete-network. This approach has been tested, e.g., in Publication III. One limitation of the policy is that it cannot be used for cases, where simultaneous availability of multiple input events is required, such as the combining of two input streams. This condition was observed with EPA7 (combining) in Publication III, p. 9.
To complement the methods available on the platforms, two new approaches to memory handling on SPARQL query networks were constructed and tested in the publications:

1. **Cleanup-rules**, which compare the timestamps or any other increasing indices of the events in an event channel and remove everything older than the latest event in each channel. Such cleanup-rules were utilised and tested in Publication I and Publication III⁹.

2. **Event-based memory handling**, which separates background knowledge and any materialisations based on the ontology into a separate named graph, and deletes each incoming event object and any resulting materialisations after the event has been processed. This approach was especially targeted for use with a materialising reasoner (Section 7), tested in Publication V, where the solution is confirmed to both produce correct results and significantly decrease memory consumption, as seen in PV Figure 11.

As indicated in Publication IV Figure 8, the memory consumption of the current INSTANS implementation is not entirely stable, but the mechanisms utilised in the experiments were shown to be effective and help to stabilise memory consumption in all the tests, where it was possible¹⁰.

Based on the cleanup-rules and event-based memory handling approaches it would be straightforward to extend the SPARQL query network implementation to a more comprehensive time-to-live mechanism of an event object supporting longer, fixed expiration values and also make the mechanism specific to the event object type. Integrating the same mechanism to the INSTANS platform would most likely lead to a higher-performance solution than SPARQL rule networks, but at the same time require support for a number of new concepts:

- **Clock**: A platform clock reference would be required. In the embedded publications the clock is implemented using a single RDF triple, updated by SPARQL rules and accessible by any query. The current version of the INSTANS platform is not time-aware.

---


¹⁰The purpose of Publication IV experiment 4 was specifically to stress memory. Consequently memory consumption in experiment 4 is not stable.
Event stream processing

- **Clock input from stream**: To maintain the capability to track any timestamps available in the input stream, the system would need the capability to find the correct attribute from incoming events. The most likely method of implementation would be a SPARQL extension.

- **Event objects in Rete**: Currently the concept of event objects only exists in SPARQL queries and rules. To be able to set a common time-to-live on the platform, the platform should be aware of the triples that compose an event object.

- **Time-to-live abstraction**: A way to define the lifetime for event object types.

Esper supports all of these concepts (declaration of event types, a global clock with an option for external synchronisation, windows defining the lifetime of events). Similar extensions would be a potential way to further develop INSTANS.
7. Reasoning on events

“All our knowledge begins with sense, proceeds hence to understanding, and ends with reason, beyond which nothing higher can be discovered in the human mind for elaborating the matter of intuition and subjecting it to the highest unity of thought.”

— Immanuel Kant, *The Critique of Pure Reason, 1781*

Reasoning is the capability to generate new knowledge out of facts and rules. For example, an inverse rule can state that for an “inverse” predicate the subject and object can be reversed (prp-inv1 from [135]). If we then have knowledge that child is the inverse of parent, receive information that John is a parent of Jim and apply the rule, we can infer that Jim is a child of John, even if we never explicitly received that fact. Support for reasoning has been built into the core of Semantic Web technologies. The ability to reason about streamed events is an important motivator for applying Semantic Web methods to event processing.

Reasoning is explored in Publication V, where the starting point was the SPARQL 1.1 Entailment Regimes recommendation [63]. The INSTANS platform does not contain a built-in reasoning tool. However, Publication V demonstrates that the rule network approach of INSTANS can be used for rule-based reasoning, translating the rules of entailment regimes to SPARQL Update. Based on the input, these rules create inferences through *materialisation* [132] of new triples, which automatically and asynchronously become input for other rules.

The tested entailment regimes are listed in Table 7.1. There are significant overlaps between the rules of the different regimes, as illustrated in Figure 1 of Publication V. Apart from $D^*$, $P$ [126, 125] and SWCLOS1 [77], which are intentionally disjoint, there is no modularity between en-

---

[Accessed Oct 5th 2016]
Reasoning on events

tailment regimes and they should not be used in parallel. Disregarding minor differences in rule input filtering\(^2\), all the tested entailment regimes together contain 100 unique rules and 21 unique unsatisfiability conditions. Publication V explains the implementation of each different

<table>
<thead>
<tr>
<th>Entailment Regime</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>RDF 1.1 semantics of RDF entailment</td>
<td>[142] 8.1.1</td>
</tr>
<tr>
<td>RDFS entailment</td>
<td>[142] 9.2.1</td>
</tr>
<tr>
<td>OWL 2 RL profile</td>
<td>[135] Tables 4-9</td>
</tr>
<tr>
<td>$\rho$df</td>
<td>[97] Table 1</td>
</tr>
<tr>
<td>D*- and P-entailment</td>
<td>[126] Tables 4 and 7</td>
</tr>
<tr>
<td>SWCLOS2 additional rules</td>
<td>[77] Tables 13.4 and 13.5</td>
</tr>
</tbody>
</table>

Table 7.1. Entailment regimes tested on INSTANS

type of rule, with the complete set of rule implementations available in the complementary web resources. It is observed that over 84% of all rules and unsatisfiability conditions have a simple and straightforward translation to SPARQL Update rules. The most common difference between SPARQL and rule semantics is the assumption that two distinct variables in rules do not assume equal values, whereas in SPARQL the following triple patterns:

```
?x a owl:Class
?y a owl:Class
```

would result in identical sets of solutions for ?x and ?y. Such cases are solved by using filters to require that the values are not the same, as shown in the rdfs7 rule example of Publication V Figure 2.

The remaining rules typically require iteration through ordered lists of elements. While SPARQL property paths can be used to match the elements of a list to a variable [137], there is no specified way at the time of writing to iterate through those elements. Rules requiring list processing are implemented either by using a generic query network for list processing (Publication V Figure 3) or as specialised list processing query networks for cases not solvable with the generic approach.

The implemented entailment regimes were conformance-tested using

---

\(^2\)Counting two rules with identical input and output triples the same, even if the corresponding entailment regimes have differences in the definitions of the pre-conditions for which input qualifies for an individual rule.
the entailment-category tests of SPARQL 1.1 Test Suite\textsuperscript{3}. The coverage of the test set in the current form was found to be both spotty (covering 17-47\% of rules depending on the regime, Publication V Table 5) and uneven (same rule may be tested up to 7 times while others are ignored). The implemented entailment regimes were found to correctly process all the tests, on which they were applicable. Additional custom rules were generated for the tests not passable by the implemented regimes, raising the INSTANS pass-rate to 89.4\% of the tests.

It was observed that only up to four rules were required for passing any single entailment conformance test. At the same time many rules produce significant overhead and are often simply ignored by implementations. Therefore the main argument of Publication V, that an entailment regime implementation should be flexible and open to the end user, was found correct. The solution of Publication V uses the same language for rule implementation as an end-user would use for writing queries – SPARQL. Not only can rules be freely selected from the provided sets or manually added, they can also be packaged together with the queries even into the same query files. Compared to a setting, where the reasoner is provided as a part of the event processing system, the separation of rules from the platform guarantees the repeatability of older experiments with the same set of rules.

As observed by Pérez-Urbina et al. [103], optimal conditions for materialisation are a stable ontology combined with stable data. Infinite, time-varying data streams are constantly changing and the resulting materialisations eventually fill any memory or database, if they have to be stored forever. However, in most applications there is a maximum timespan after which old data – and the generated materialisations – can no longer impact the result. In some cases this is a time window [15], in other cases a single event is enough. Publication V Figure 6 illustrates a solution, where static background knowledge is first moved into a named graph. Event data is then processed in the main graph and deleted – together with the associated materialisations – after each event (ref. Section 6.6).

\textsuperscript{3}http://www.w3.org/2009/sparql/docs/tests/ [Accessed Mar 26th 2016]
Reasoning on events
8. Performance

“Amid the pressure of great events, a general principle gives no help.”

– Georg Wilhelm Friedrich Hegel, *Philosophy of History, 1837*

Understanding the expected performance of different stream processing platforms is important for pre-evaluating their suitability for a given task. It is relatively common for stream processing benchmarks to permit different sets of answers to the same query. The differences are typically due to one or more of these five aspects:

1. *Input order:* Some test arrangements may introduce random differences in the data order of arrival; e.g., in [12] it is stated that the results may depend on the order of tuples in the stream.

2. *Random input:* The process may involve a random element, which impacts query results under different executions. In [12] a simulator generates road accidents in random locations and these accidents influence the road toll for the impacted segment.

3. *Real-time timestamp assignment:* In some test arrangements the timestamp, which is used as the basis for time window processing, is derived from the clock of the computing hardware and attached to the event by the stream processing platform. This introduces random variations to the timestamps depending on differences between execution environments and even for a single environment due to any of a multitude of operating system tasks (e.g., I/O, memory handling). Differences in timestamps result in differences in assigning input data to time windows, ultimately leading to different query answers.
4. *Missing synchronisation and/or flow control:* In practical use cases a stream producer typically broadcasts the data stream to multiple recipients. In such scenarios it is not feasible to flow control the output based on the ability of an individual receiving entity to digest the data in time. In many test arrangements the input stream and stream processing platform are connected without flow control on the interface. In [82] both duplicate answers due to the stream processing platform executing multiple times on the same data as well as missing answers due to input data being overwritten during high load are observed.

5. *Different operational semantics:* As there has been no commonly specified stream query language with agreed operational semantics for RDF, different platforms use different interpretations. For RDF DSMS platforms this aspect has been addressed especially in [47]. Work for a unified RSP Query Language (QL) is ongoing [48, 49], but not available in platforms at the time of writing.

As the INSTANS platform was developed alongside the experiments in the attached publications, focus was on repeatable, verifiable experiments. All the experiments in the included publications are based on test arrangements without any random elements:

- **File input without parallelism:** All test streams were saved onto disk files and all experiments involved input from only one file at a time. This guarantees that the order of events is the same for all executions also when a file contains different types of events from different sources, which in other environments could be described as separate streams.

- **Clock synchronisation to the stream:** In all tests of Publication I, Publication III and Publication IV all timestamps were incorporated into the events. Using explicit timestamps from the stream guarantees identical processing regardless of any differences in processing speed.

- **Asynchronous stream input:** All streams were read and processed asynchronously on all the involved platforms without loss or duplication of data. This permitted measurements of the average speed of execution during a given test batch. As the speed of execution during the batch may vary, the average speed is indicative of, but not a fully accurate
characterisation of the achievable maximum constant processing rate.

- **Confirmed query answers:** In Publication I and Publication III the correct answers were confirmed manually. The only differences in answers in any of the experiments were in Publication I between INSTANS and C-SPARQL due to the use of non-overlapping stream windows on C-SPARQL, which masked the detection of some event pairs. In Publication IV answers to shorter test batches were checked manually. For all batches INSTANS and Esper were confirmed to produce identical output. In Publication V the LUBM output of all tested platforms was confirmed to match the sample output available in the LUBM material.

### 8.1 Summary of INSTANS performance tests

The performance of the INSTANS platform was tested in Publications I, III, IV and V. Even though the name of the platform is the same, the tests for Publication I were carried out using a Scala-based implementation by Haris Abdullah running on the Java Virtual Machine (JVM) (“Scala-Rete” v. 0.1), whereas all other tests were performed on a completely separate Common Lisp implementation by Esko Nuutila (“CL-INSTANS”). While the detailed performance results can be found in each of the publications, a high-level summary is included here.

The Publication I sample application – “Close Friends” – combined background knowledge of friendship connections with timestamped location updates from individuals to yield indications of nearby friends. The datasets were generated by a simulator with subjects moving around a real map surface [109] obtained from OpenStreetMap\(^1\). The Stream Social network data Generator (S2Gen) in LSBench [82] resembles the position aspect of Close friends, as it uses also a stream of the GPS locations of users. Comparisons were made with C-SPARQL, but since the task was targeted for event processing rather than data stream processing, there were some challenges in the conversion of the experiment to C-SPARQL. In order to keep multi-triple events together despite the stream of timestamped triples used in C-SPARQL, windows based on the number of triples rather than timestamps had to be used. With triple-based windows no window overlap was possible, resulting in some missed detections.

\(^1\)https://www.openstreetmap.org/ [Accessed Mar 22nd 2016]
of event pairs across window borders. Had overlapping windows been possible, duplicate detections would have been created. The main parameter for measurement was the notification delay from the availability of the last of the pair of matching location events to the time when a nearby notification was issued. The Scala-based INSTANS v. 0.1 responded in about 12 ms on the test platform, whereas the C-SPARQL notification delay was dominated by the chosen window length, e.g., 1.34 seconds for a 5 second window.

Publication III includes performance tests of individual event processing agents as well as a combination of all the eight EPA presented in the paper. The INSTANS built-in remove-policy was tested to be faster than a SPARQL-encoded cleanup-rule in all other cases except EPA4 (project). The explicit cleanup-rule was 1.1 times faster for EPA4, the slowdown factors for other EPAs were measured as 1.29-1.66. The throughput of individual EPAs varied depending on the complexity with EPA8 (pattern recognition) being the slowest at 736 events per second (eps) and EPA6 (aggregation) being the fastest at 1,810 eps, which would translate to approximately 11,000 triples per second (Table 8.1). A federated SERVICE query was also tested over the internet, yielding a speed of 3.7 eps compared to 545 eps, when binding the corresponding values locally. The complete network of eight EPAs ran at 176 eps. Taking into account that the filters in the complete network prevent all events from reaching all the EPAs, the sum of the times running each EPA separately, 134 eps, is closer than would be intuitively expected and indicates that the increased complexity in the combined Rete-network slows down processing. Since there was no other connectivity needed between the EPAs than the designated event channels, it is expected to be beneficial to execute each EPA in an isolated Rete environment.

Due to difficulties in finding a comparison event processing platform from the RDF domain, Publication IV takes a step back and searches for a comparison platform outside the RDF and SPARQL domains. Unfortunately at the time of selecting trial platforms most CEP systems were commercial products, requiring paid licenses. Additionally, the licensing terms in many products forbid publication of performance results. Esper emerged as a good candidate with open-source code and a suitable licensing policy. The challenge was that INSTANS and Esper shared neither common input data format, nor a common query language. A conversion of input data to a stream of XML documents was provided for Esper, and
the task – supply chain logistics monitoring – was converted to EPL, the proprietary language supported by Esper. Esper community version was distributed as a Java library, and a new Scala-application XEvePro\textsuperscript{2} was created to handle file input, timestamp conversion, result comparison and execution time measurement. Whereas INSTANS is a continuously evaluating rule network, Esper is driven by explicitly defined events. Due to these different approaches the resulting stream processing applications in SPARQL and EPL didn’t have exactly the same operating algorithms, but empirical testing confirmed that the results were identical. While the tests showed that Esper was clearly faster in all tested cases, INSTANS was also shown to run 950 times faster on the test laptop than the target derived from real-world manufacturing.

Publication V concentrated on reasoning, which was addressed in more detail in Section 7. Performance tests were carried out using the LUBM benchmark [66] in order to incorporate non-streaming RDF processors into the comparison. Other RDF stream processing platforms could not be used for comparison of the implemented entailment regimes, as the available stream reasoning support on SparkWave, ETALIS and C-SPARQL was limited to custom subsets of RDFS entailment, while LUBM required rules from OWL 2 RL [135].

The main target was to demonstrate the importance of an open reasoning implementation in enabling the user to select only the required rules. This was clearly accomplished: a complete set of OWL 2 RL rules without extra measures for memory handling did not complete any query on INSTANS. Using only the required rules with no memory handling optimisations caused many queries to collapse between set sizes of 5 and 10 universities. Combining optimised rule sets with event-based memory handling completed all LUBM queries requiring reasoning up to 100 universities without problems. Performance was also compared with Jena\textsuperscript{3}, as it has a materialisation-based reasoner and Stardog\textsuperscript{4}, which has a fast query-rewriting reasoner. It was confirmed that INSTANS, Jena and Stardog produced exact matches of the LUBM sample results\textsuperscript{5}. Jena was faster than INSTANS with the queries using more limited rule frameworks, but with the most complete OWL reasoner it also had difficulties

\textsuperscript{2}https://github.com/aaltodsg/xevepro [Accessed Sep 22nd 2016]
\textsuperscript{3}https://jena.apache.org/ [Accessed Mar 29th 2016]
\textsuperscript{4}http://stardog.com/ [Accessed Mar 29th 2016]
\textsuperscript{5}http://swat.cse.lehigh.edu/projects/lubm/answers.htm [Accessed Mar 29th 2016]
completing any tests, further underlining the importance of optimised configuration of rules. Stardog performed 9-364x faster than INSTANS, but the version under testing had problems completing three (out of 14) queries with a test set of 100 universities.

The common theme in all the included performance tests became the challenge of finding a platform and a scenario for a true apples-to-apples comparison with INSTANS. The platforms available in the RDF stream processing domain have concentrated on the DSMS paradigm, which – as discussed earlier – targets a different domain of problems, dealing with the calculation of aggregate values over time windows. While the calculation of aggregates can also be done with INSTANS\(^6\), the platform is not optimised for such tasks\(^7\), and all the available example tasks of interest have required processing of exact patterns of events rather than statistical aggregate calculations. All tests on INSTANS have been based on the processing of an asynchronous input at maximum speed, allowing to measure the maximum average rate of processing, but at the same time producing correct answers without any ambiguity. While this allows to confirm that a query answer is correct, it does not fully address the capability of the platform to keep up with a synchronous data source throughout the processing of the stream, as it only gives an average result for the processing of the complete input data batch.

As pointed out in, e.g., [12], the proper performance measure for a stream processor would be the time difference from the arrival of input to the time when output is available, a.k.a. response time or notification delay. This was only measured for Publication I. In half of the scenarios of Publication IV the output was based on a timeout after the availability of one of a pair of events, in which case the response time after an incoming event would be less important. Especially in the scenario of Publication V, where the performance was compared with non-streaming processors, Jena and Stardog execute the query as a batch over the complete dataset and no accurate reference point is available for measuring query response time during processing.


\(^7\)A set of SPARQL rules need to be used as in Publication III Figure 10 instead of specialised language extensions.


8.2 **INSTANS throughput comparison**

The performance tests executed over the INSTANS platform in the embedded publications have a number of differences: versions of the INSTANS platform, execution hardware, size and format of events as well as complexity of the task, including the need for buffering data. But as the amount of triples can be counted for each dataset, and the total time of execution has been measured for each experiment, the absolute performance as triples per second can be computed over all the experiments. Selected example results sorted from slowest to fastest are listed in Table 8.1. The tests, hardware (HW) and software (SW) versions in the table are the following:

- **Close Friends**: notifications of nearby friends from Publication I.

- **CEP2SPARQL**: *Complex Event Processing* agents as shown in Publication III.

- **Supply Chain**: pharmaceutical manufacturing monitoring performance from Publication IV.

- **LUBM**: *Lehigh University Benchmark* as tested in Publication V.

- **Old HW**: MacBook Pro with 2.26 GHz Intel Core 2 Duo, 8 GB 1067 MHz DDR3 memory running OS X version 10.9.5.

- **New HW**: MacBook Pro with 2.7 GHz Intel Core i5, 16 GB 1867 MHz DDR3 memory running OS X version 10.10.5.

- **Scala-Rete**: INSTANS v. 0.1 coded in Scala, executed over Scala v. 2.9.1-1 running over Java 1.7.0_72.

- **CL-INSTANS**: V. 0.3.0.0\(^8\) coded in Common Lisp, executed over SteelBank Common Lisp (SBCL).

The performance results of the *Close Friends* example did not fit into the original Publication I due to space constraints, but they have been re-
leased as a separate technical report [107]. The performance of the Close Friends example executed over CL-INSTANS has not been published before. To ensure alignment of Java and OS versions, also the Close Friends Scala-Rete results were re-executed for Table 8.1. The Supply Chain results reported in Publication IV as Electronic Product Code (EPC) per second have been scaled to events/s and triples/s for Table 8.1. The LUBM data does not define an event, so the performance in terms of events per second has not been calculated for the LUBM examples.

Table 8.1 shows that the new CL-INSTANS is 6.7x faster than the Scala-Rete implementation on the Close Friends example on the same (old) HW, whereas the impact of the new HW with CL-INSTANS is 1.7x. The highest performance of over 21 ktriples/s is achieved on Q1 of LUBM, which is a straightforward query and requires neither reasoning nor buffering. The slowest examples due to buffering in Publication IV have been omitted from the summary, as they were not completed. The processing speed in terms of events/s roughly aligns with triples/s apart from the Supply Chain examples (less events/s than other cases with similar triples/s performance). This result is expected, because the events in Publication IV summarise 20 or 100 EPC, depending on event type, and are therefore far larger and involve more computation per event than the single location events used in Close Friends or CEP2SPARQL.

Table 8.1. Relative performance of selected tests from the included publications, sorted by triples per second.

<table>
<thead>
<tr>
<th>Test</th>
<th>HW</th>
<th>SW</th>
<th>Case</th>
<th>Events/s</th>
<th>Triples/s</th>
</tr>
</thead>
<tbody>
<tr>
<td>Close Friends</td>
<td>Old</td>
<td>Scala-Rete</td>
<td>10,000 events</td>
<td>59.1</td>
<td>532</td>
</tr>
<tr>
<td>CEP2SPARQL</td>
<td>New</td>
<td>CL-INSTANS</td>
<td>EPA-All 10,000 events</td>
<td>176</td>
<td>1,053</td>
</tr>
<tr>
<td>LUBM</td>
<td>New</td>
<td>CL-INSTANS</td>
<td>Q5 10U opt event</td>
<td>2,513</td>
<td></td>
</tr>
<tr>
<td>Close Friends</td>
<td>Old</td>
<td>CL-INSTANS</td>
<td>10,000 events</td>
<td>394</td>
<td>3,548</td>
</tr>
<tr>
<td>CEP2SPARQL</td>
<td>New</td>
<td>CL-INSTANS</td>
<td>EPA8 10,000 events</td>
<td>736</td>
<td>4,415</td>
</tr>
<tr>
<td>LUBM</td>
<td>New</td>
<td>CL-INSTANS</td>
<td>Q11 10U opt event</td>
<td>4,439</td>
<td></td>
</tr>
<tr>
<td>Close Friends</td>
<td>New</td>
<td>CL-INSTANS</td>
<td>10,000 events</td>
<td>670</td>
<td>6,029</td>
</tr>
<tr>
<td>Supply Chain</td>
<td>New</td>
<td>CL-INSTANS</td>
<td>Exp1 1M events</td>
<td>151</td>
<td>7,291</td>
</tr>
<tr>
<td>Supply Chain</td>
<td>New</td>
<td>CL-INSTANS</td>
<td>Exp5 100k events</td>
<td>164</td>
<td>7,918</td>
</tr>
<tr>
<td>CEP2SPARQL</td>
<td>New</td>
<td>CL-INSTANS</td>
<td>EPA6 10,000 events</td>
<td>1,810</td>
<td>10,858</td>
</tr>
<tr>
<td>LUBM</td>
<td>New</td>
<td>CL-INSTANS</td>
<td>Q1 10U static</td>
<td></td>
<td>21,195</td>
</tr>
</tbody>
</table>

While the figures shown in Table 8.1 are indicative of the performance achieved with current laptop HW, it must be remembered that many possibilities for performance improvement have not yet been explored in these tests:
- **Multi-core processor architectures**: As observed in Publication IV, the INSTANS implementation over SBCL is currently only able to use a single processor core.

- **Modular Rete-engine**: The current tests were executed with all the active rules compiled into a single Rete-engine. Especially in cases like EPA-All of Publication III complexity and unnecessary verifications would be reduced if each EPA was run in a separate and independent Rete-module.

- **Parallel HW**: In addition to Rete-engine modularity, many of the tasks would permit manual parallelisation and execution on multiple computers. As INSTANS uses RDF for both input and output, tasks can be executed both serially and in parallel.

- **Indexing of comparisons**: Especially in Experiment 4 of Publication IV a dramatic impact of event buffering on performance can be observed. The issue is mostly caused by a rule which checks, whether any of the events in the buffer would need to be reported as an anomaly. This rule is re-checked for all the buffered events every time when time moves forward, which takes place with every new event arrival. Based on preliminary testing after Publication IV, a very significant performance improvement was achieved by indexing those comparisons using the time attribute, after which time only needs to be compared at the end (min or max) of the index.

Publication IV contains comparisons with Esper. As Esper does not support RDF, throughput can only be calculated as events/s. Esper was measured to process 2,355 events/s for Experiment 1 1M event batch (15.6 times INSTANS) and 274 events/s for Experiment 5 100k (1.7 times INSTANS). Publication V includes throughput comparisons with Jena and Stardog. Stardog was measured to run Q11 10U at 197,789 triples/s (44.6 times INSTANS). Jena did not complete 10U, but 5U was measured as 71,785 triples/s (16.2 times INSTANS 10U throughput).

---

9Publication IV Section 4.3 steps 3 and 4 for INSTANS
10Improvement for the 100,000 EPC case was 86 times faster, from 20 EPC/s to 1,736 EPC/s.
8.3 Observations on the performance of INSTANS

Based on the numeric results available (Section 2.8.1), INSTANS results are fairly well positioned among other results obtained from RDF stream processing platforms. However, both non-streaming RDF platforms and non-RDF stream processing platforms demonstrate much faster processing speeds. The optimisation techniques available to non-streaming platforms are different from stream processing and explain a part of the difference, complemented by the longer development cycle in tougher competitive landscape (more candidates) leading to higher maturity for the top platforms. For non-RDF stream processing platforms the driving force has been the extensive commercial use, forcing the top platforms to push the state-of-the-art in parallel processing, leading to very high performance on server clusters. At the same time the parsing and processing of a fixed-column CSV-format table row or a fixed JSON object is far easier to optimise than that of a free-form RDF graph, for which no assumptions on similarity of subsequent events can be made. An event in a uniform stream can be directly assigned to a programming-language object, whereas every RDF graph event has to be separately and dynamically parsed without any a priori assumptions on the format of the resulting object. There clearly is a computational price to pay both for “heterogeneous events” and “layered stream processing”. To fully benefit from using a platform like INSTANS, these aspects should be a part of the task.

The performance results listed in Section 2.8.1 are indicative of the performance differences between the event processing platforms with a permissible license towards result publication. The platforms based on Semantic Web formats with available performance comparisons against a non-Semantic Web based platform, INSTANS and ETALIS, are programmed in Common Lisp and Prolog, respectively. No other referenced platform is using either one of these programming languages. Support for reasoning makes rule-based languages with an artificial intelligence focus particularly interesting for Semantic Web applications. Most of the non-Semantic Web based platforms are implemented in a language from the Java family. The non-Semantic Web platforms typically do not support reasoning, allowing simpler and faster-performing implementations. Consequently no performance comparisons for event processing tasks requiring reasoning can be carried out.

The efficiency of a platform for a given task can be roughly estimated
from the currently available results. However, these results cannot be extended to state that either Semantic Web based solutions or non-Semantic Web based solutions would be more performant, because performance comparison between two platforms can only be carried out for the subset of capabilities supported by both platforms. Semantic Web support for, e.g., fully flexible event formats, globally accessible ontologies and reasoning may all cause processing overhead, but they also enable a wider scale of applications.
9. Discussion

“If we knew what it was we were doing, it would not be called research, would it?”

– Albert Einstein

The high-level goal of this study was to investigate how event processing could be carried out using the existing Semantic Web technologies. The increasing interest for event processing originates from the quickly increasing availability of online sensor input, including people and their mobile devices, environmental infrastructure sensors and industrial RFID readers. Once data to make an informed decision exists, the winner is the one who is fastest in converting input into action. The drivers for the application of Semantic Web in this context are the support for generic event object structures, the commonly agreed specifications, the globally accessible ontologies with tools to cope also with differences and discrepancies to glue together event types in a multi-supplier environment, the support for online fusion with existing background knowledge both locally and through federated queries, the possibility for delayed semantic commitment and the support for reasoning, which can be used to infer new knowledge online and make integration in a multi-vendor environment easier. Further qualitative analysis of the benefits of Semantic Web technologies compared to other approaches is provided in Publication IV.

When this study began, proposals for language extensions in the SPARQL stream processing space were available. For many of the proposed extensions, an execution platform existed. Nearly all of the available extensions and platforms followed the data stream processing paradigm, in which the first operation – stream-to-relation – is to extract a time-bounded segment – a time window – from the stream, after which persistent relational queries are repeatedly executed for each time window as the stream progresses. One platform [9] provided sequence opera-
tors to search for event sequences.

The essence of complex event processing as laid out in [88], however, is the support for hierarchical processing, where lower-level simple events are summarised as higher-level composite and complex events to provide more tangible information. Even though complex event processing is sometimes understood as the handling of event patterns, in the classification of event processing agent types in [53] the recognition of event patterns is only one agent type out of a set of eight. None of the RDF stream processing platforms available at the start of this study supported a flexible way to continuously and asynchronously execute query networks of arbitrary sizes, which could implement event processing networks, supporting modular use of event processing agents.

The event processing paradigm was found to be a bad match with data stream processing, because the extraction of stream-level time windows without synchronisation to event objects may break event patterns. Defining large-enough windows with sufficient overlap, a target pattern can be captured by a correctly positioned window, but at the cost of maintaining a large window buffer, high computational overhead from repeated processing of both wanted and unwanted events, filtering of duplicate solutions and filtering of false solutions. False solutions can be generated when an event pattern crosses a window border, especially in cases where event absence is monitored, as illustrated in Publication IV Figure 7.

From these starting conditions the main task became the creation of methods for the SPARQL query language to build hierarchical event processing networks for heterogeneous structured events. Several missing pieces had to be filled in. Stream processing in RDF was originally done on triples with externally encoded timestamps (TRDF), which did not properly support more structured events containing multiple attributes. The first step was the integration of timestamps inside event RDF in Publication I, extended in the vocabulary of Publication II to cover three timestamps measured from different points of reference. The choice of timestamp was implemented into the time processing SPARQL queries (Publication IV). The solution remains syntactically compatible with all pre-existing RDF specifications and tools.

Turtle-serialised events were used in Publication I and Publication II. Problems with event object punctuation became apparent in the preparation of Publication III, at which point the recently released TriG specification [26] was quickly adopted to unambiguously encapsulate event objects.
The event-related ontologies available at the time concretised the semantic ambiguity of the word *event* in addressing everything from concerts to samples from agricultural sensors. The missing constructs for separate header and body segments, composite and complex events as well as multiple timestamps depending on the point of measurement were specified in Publication II as a modular *event processing ontology design pattern*, aligned with other important ontologies in the domain.

In the beginning of the study SPARQL Update was being finalised as a dataset maintenance language, not as a rule language for continuously processing rule networks. Publication I demonstrated how SPARQL Update could be used to create an event processing network, using named graphs as event channels between the SPARQL-encoded event processing agents and rule-edited triples for persistent storage. In addition to the ontology design pattern, Publication II also showed how operations on complex and composite events can be carried out with SPARQL in a modular way as well as explored current modularity restrictions in querying. Publication III extended the approach of Publication I to cover all types of event processing agents found in related literature [53, 123]. Especially the computation of aggregate values and recognition of event patterns as SPARQL rule networks were demonstrated for the first time in [108], a predecessor of Publication III.

As the complexity of both individual agents and the event processing network increased, observations on potential improvements to SPARQL syntax were made. Even though dataset input as TriG or NQuad serialisations was possible, there was no syntax to generate comparable TriG output\(^1\). Additionally, combinations of SPARQL Query and Update clauses, which are not permitted by the current grammar, would have reduced unnecessary repetition and simplified important operations. Apart from the minor modification to support dataset output and an extension function to convert XSD timestamps to integer values, no language extensions were used to produce any of the included results. It would also have been possible to use integer format timestamps in the streams, in which case even that conversion function would have been unnecessary.

The envelope for SPARQL rule networks was further pushed in Publication V by implementing generic user-controllable support for reason-

\(^1\) INSERT only carries out memory-internal graph writing operations on the INSTANS platform.
ing. Multiple entailment regimes were implemented and conformance-tested with the SPARQL 1.1 Test Suite. Through the addition of custom rules beyond the available entailment regimes, conformance test results in line with the highest results reported for other platforms were demonstrated. Publication V also demonstrates event-based memory handling implemented as a SPARQL query network, separating static background knowledge from event-based data and the related materialisations through the use of named graphs, decreasing memory consumption. Benchmark queries, which crashed without event-based memory handling, completed ten times larger datasets successfully with the event-based memory handling activated.

All presented solutions were verified on the INSTANS platform undergoing simultaneous development. It was a unique tool due to the continuous asynchronous processing of SPARQL query networks over blocks of RDF, different from repetition of persistent queries over time windows of timestamped RDF triples utilised by other RDF stream processing tools.

The approach to use standard non-extended SPARQL led to a lower-level approach to coding than on platforms with specific stream processing extensions. Processing of timestamps using SPARQL query networks instead of built-in platform support gives the flexibility of matching a task-specific timestamp attribute. With built-in handling of timestamps a solution for the platform to find the correct timestamp attribute has to be provided in order to benefit from the same liberty. One solution would be to provide a formally specified stream description, which would accurately describe the timestamps available in the stream and their reliability (e.g., accuracy and optionality).

Due to the unique characteristics of the approach, exceptional effort had to be invested into performance comparison. The example application of Publication I was kept on a level of simplicity enabling execution on a data stream processing platform. The comparison remains somewhat artificial, as the stream handling approach of the data stream processing platform did not permit the use of overlapping windows on multi-triple events. As a result, the tumbling window segmentation of pairs of independent events misses some solutions on the comparison platform. Also notification time, which was the main parameter under study, becomes primarily dependent on the length of the time window on the comparison platform and not the actual execution speed of the platform.

As Publication III targeted the implementation of all eight types of EPA
presented in literature [53] using structured heterogeneous events, RDF and SPARQL, performance could only be demonstrated using INSTANS. To enable cross-platform comparison, Publication IV gave up on both RDF and SPARQL. A concrete event processing task from pharmaceutical manufacturing was programmed and executed on both INSTANS and an event processing platform using XML data and EPL queries. Despite different event formats, query languages and operating principles leading to differences in query algorithms, identical results were verified for both implementations. Publication V concludes the performance studies by confirming that also sets of >100 simultaneous interconnected rules can be successfully executed on INSTANS, as well as demonstrating the benefits of end-user customisation of the reasoning rule set in terms of repeatability, processing speed, stability and memory consumption. As no RDF stream processing platforms with support for the necessary entailment regimes were available, comparisons in Publication V were made against non-streaming RDF processing platforms using a non-streaming benchmark.

The publications incorporated in this volume and the experiments provided in the referenced network resources provide a clear answer to the original question: RDF and SPARQL can be used for hierarchical processing of layered heterogeneous events. All of the event processing tasks under study in the different publications have been successfully carried out and the results verified. To achieve this result, multiple novel methods of applying existing building blocks and connecting query language constructs had to be developed. The data model remains fully compatible with existing RDF and TriG specifications.

At the same time it is concluded that current SPARQL Query and Update definitions do not provide an ideal environment for building event processing applications. This is hardly surprising, because they were created for a different purpose. Missing combinations of query and update operations as well as the lack of a mechanism for creating macros result in unnecessary coding overhead. Lack of rule priorities result in more complex debugging scenarios and the need for different execution policies for different kinds of event processing agents.

The performance of the current INSTANS implementation on a present-day laptop was shown to meet expectations for all of the applications that the platform was considered for. At the same time it could not compete with a specialised event processing platform. At this time there are no
tools to quantify how much of the observed difference is due to the efficiency of the current programming environment, and how much is due to complexity resulting from the chosen approach, such as the schema-less heterogeneous RDF graphs, requiring event objects to be processed triple-by-triple instead of being able to directly assign them to schema-based objects, or the Rete rule-network paradigm in general, which may result in a lot of processing overhead in cases where a lot of filter conditions need to be frequently re-evaluated. Based on test scenarios in Publication IV, verifications of objects in memory – also when they don’t immediately produce results – have a heavy impact on performance. Cross-connectivity could be reduced by modularising the different event processing agents, separating the corresponding Rete networks. To improve speed and decrease the likelihood of programming errors in larger event processing networks, this would clearly be an approach for future exploration. Other approaches to exploit trade-offs between flexibility and performance would be to encode events as JSON-LD (possibly with predefined schemas), simplify the query language, provide optimised extensions for event-based window processing, drop the continuously processing rule network in favour of a pipelined asynchronous network of event processing agents and further improve memory handling policies. All of these optimisations could be carried out without sacrificing the key element of a distributed system with support for global concepts.

Even though this body of work has demonstrated that RDF and SPARQL can serve as the basis of a complex event processing system with reasoning support, the demand for such a solution may appear limited at this time. Most stream processing currently takes place in confined environments, or at least under the directives defined by the stream publisher, where the support for either globally addressable ontologies or reasoning based on a generic set of rules is unnecessary. Situations where streams from multiple sources need to be combined are sufficiently rare that tailored solutions remain feasible. Most event streams are fairly homogeneous, in which case object encoding such as JSON or even CSV allow for easier optimisation of program code to parse incoming events than freeform RDF.

Explosive growth in the quantity of real-time data, wide availability of data streams from multiple suppliers with multiple event types and different formats, as well as the need to complement event data with federated queries and implicit rule-based knowledge are all drivers in favour of ap-
plying the investigated methods. When current approaches fall short in handling the growing web of event streams, the methods studied in this dissertation can be used to address those issues.
References


Availability of data streams sourced from transactions and networked sensors is experiencing explosive growth. As more actors join in, the processing of event streams from highly distributed and diverse sources will become a major challenge. This book demonstrates how the Semantic Web metamodel RDF and the SPARQL query language can be used for layered processing of patterns of structured events. Reasoning and federated queries, which belong to important building blocks of the Semantic Web, are transferred to an event stream environment. The solutions in this volume -- all tested in practice -- can jointly build comprehensive event processing applications ready to meet the challenges of a networked online future.