N°d'ordre NNT :

THESE de DOCTORAT DE L'UNIVERSITE DE LYON
opérée au sein de
Université Jean Monnet – SAINT-ETIENNE

Ecole Doctorale N° ED 488-SIS
Sciences Ingénierie, Santé

Spécialité de doctorat :
Discipline : Informatique

Soutenue publiquement/à huis clos le 04/11/2016, par :
Syed Gillani

Semantically-enabled Stream Processing and Complex Event Processing Over RDF Graph Streams

---

Devant le jury composé de :

Bonifati, Angela	Professeur Université de Lyon, France	Présidente
Bonifati, Angela	Professeur Université de Lyon, France	Rapporteure
Rousset, Marie-Christine	Professeur Université Grenoble Alpes, France	Rapporteure
Mileo, Alessandra	Adjunct Lecturer INSIGHT NUI Galway, Ireland	Examinatrice
Lecue, Freddy	Principal Scientist Accenture Tecnology Labs, Ireland	Examinateur
Laforest, Frédérique	Professeur Université Jean Monnet	Directrice de thèse
Picard, Gauthier	Maitre Assistant HDR Ecole des Mines Saint Etienne, France	Co-directeur de thèse

To my parents and sisters

Abstract

There is a paradigm shift in the nature and processing means of today's data: data are used to being mostly static and stored in large databases to be queried. Today, with the advent of new applications and means of collecting data, most applications on the Web and in enterprises produce data in a continuous manner under the form of streams. Thus, the users of these applications expect to process a large volume of data with fresh low latency results. This has resulted in the introduction of Data Stream Processing Systems (DSMSs) and a Complex Event Processing (CEP) paradigm – both with distinctive aims: DSMSs are mostly employed to process traditional query operators (mostly stateless), while CEP systems focus on temporal pattern matching (stateful operators) to detect changes in the data that can be thought of as events.

In the past decade or so, a number of scalable and performance intensive DSMSs and CEP systems have been proposed. Most of them, however, are based on the relational data models – which begs the question for the support of heterogeneous data sources, i.e., variety of the data. Work in RDF stream processing (RSP) systems partly addresses the challenge of variety by promoting the RDF data model. Nonetheless, challenges like volume and velocity are overlooked by existing approaches. These challenges require customised optimisations which consider RDF as a first class citizen and scale the process of continuous graph pattern matching.

To gain insights into these problems, this thesis focuses on developing scalable RDF graph stream processing, and semantically-enabled CEP systems (i.e., Semantic Complex Event Processing, SCEP). In addition to our optimised algorithmic and data structure methodologies, we also contribute to the design of a new query language for SCEP. Our contributions in these two fields are as follows:

- **RDF Graph Stream Processing**. We first propose an RDF graph stream model, where each data item/event within streams is comprised of an RDF graph (a set of RDF triples). Second, we implement customised indexing techniques and data structures to continuously process RDF graph streams in an incremental manner.

- **Semantic Complex Event Processing**. We extend the idea of RDF graph stream processing to enable SCEP over such RDF graph streams, i.e., temporal pattern matching. Our first contribution in this context is to provide a new query language that encompasses the RDF graph stream model and employs a set of expressive temporal operators such as sequencing, kleene-+, negation, optional,

conjunction, disjunction and event selection strategies. Based on this, we implement a scalable system that employs a non-deterministic finite automata model to evaluate these operators in an optimised manner.

We leverage techniques from diverse fields, such as relational query optimisations, incremental query processing, sensor and social networks in order to solve real-world problems. We have applied our proposed techniques to a wide range of real-world and synthetic datasets to extract the knowledge from RDF structured data in motion. Our experimental evaluations confirm our theoretical insights, and demonstrate the viability of our proposed methods.

Acknowledgements

The road to a successful PhD is long and labour intensive. During the last three years or so, I have experienced countless setbacks: each one plunging my hopes and self confidence to the ground. However, through these setbacks, I have undergone a genuine and powerful transformation – both intellectually and personally. The main thing I have learned from this experience is that, no matter how difficult and improbable the task is if you keep on walking in the right direction, and keep on looking for the probable remedies, you will definitely find the solution. Critically, this journey would have been unbearable were it not for the great support from the following people.

This thesis owes its existence to the help, support and inspiration from two of my great advisors: Frédérique Laforest and Gauthier Picard. Without the countless discussions with them, I would have not been able to establish the frontier works that were able to impact on the development of this research. I am greatly indebted to Frédérique for spending hours remotely (on weekends and during holidays!!) to give me advice, proofreading and correcting my "s" and "the" mistakes. Over the past three years or so she has not only helped me in writing technical papers, but has also assisted me with the tedious administrative tasks. Gauthier, on the other hand, provided me with a different view of research and offered me some critical suggestions that moulded my research. He has also been extremely supportive and understanding, especially with the choice of my research path. In addition, I would like to thank my advisors for supporting me and making it possible for me to attend numerous summer schools and conferences. No matter how much I write in this note, it is impossible to express my sincere gratitude to my advisors.

I would also like to thank all my colleagues from my group formally known as "Satin-lab": Abderrahmen Kammoun, Christophe Gravier, Julian Subercaze, Kamal Singh, Jules Chevalier. Especially to Christophe and Julien for giving me feedback and insightful suggestions to improve my work. In addition, a special thanks to Antoine Zimmerman for offering me his services and insights into the theoretical aspects of my work, and my thesis reviewing committee (Angela Bonifati and Marie-Christine Rousset) for providing insightful comments and suggestions.

Finally, thanks to my great family for giving me so much support and guidance. Mom, Dad and my sisters, you guys are the best, and you instilled in me the confidence, curiosity and discipline it takes to be successful. Thank you so much. I would also like to thank my two best mates Calum and Adam for providing me with such a great company. Last but not least, thanks to my girlfriend and Psychiatrist Céline for supporting me during my difficult times and encouraging me to be up to the task.

Il n'existe pas de chemin tracé pour mener l'homme à son salut; il doit en permanence inventer son propre chemin. Mais pour inventer, il est libre, responsable, sans excuse et tout espoir est en lui.

There is no trace out path to lead a man to his salvation; he must constantly invent his own path. But, to invent, he is free, responsible, without excuse, and every hope lies within him.

– Jean-Paul Sartre

Contents

Appendices

List of Figures

List of Tables

1 Introduction

The World Wide Web (WWW), now 27 years old, is a massive communication breakthrough and has captivated minds during the last decade or two. Tim Berners-Lee's simple creation has been transformed into a digitally connected world with unprecedented fluency in inter-communication and information dissemination. The WWW forms a connected network where the basic units being connected are pieces of information, and links that conform to the relationships join the pieces of information. At the basic level, WWW is an application to enable the distribution of information at low cost with an unlimited audience: the size of the potential audience of a shared document on the Web is limited only by the demand for it. As a result, today's web contains around 232 billion[1] unique web documents containing diverse forms of information.

However, the question is how to make sense of these information sources: humans not only need machines to store such a large repository of information, but also to query, extract, analyse, categorise and organise information for consumption. For such tasks, the starting point is to use search engines or, to use the modern cliché, to *Google* it. These (keyword-based) search engines generally employ relational-based stores (i.e., Database Management Systems, DBMSs) to store relationships between keywords and the associated Web content. However, such search engines can only direct users to the relevant reading list – usually containing a long list of documents – from which the user has to glean thereafter. The main drawbacks of keyword-based search engines include: high recall and low precision of retrieved web documents, queried results are highly sensitive to the vocabulary used by each document, and the user has to manually initiate several queries to collect the related documents. Even if the users are somewhat happy with the results, the main obstacle in providing the required support for searching the Web is that its content is not *machine readable*.

In order to address these shortcomings, the *Semantic Web* was introduced. Its primary goal is to provide machine readable content over the Web such that it can be reused and integrated with other related ones, and machines can – to some extent – interpret content on the users' behalf. During the last decade or so, the Semantic Web community has progressed by leaps and bounds. It started as a prototype research through the visionary

[1] Source: `http://www.worldwidewebsize.com/`

ideas of Tim Berners-Lee, and recently we have seen its adoption even at the industrial level. The whole consortium of Semantic Web relies on its data model called RDF and ontological languages such as RDF Schema and Web Ontology Language (OWL). RDF data consist of triples, where an RDF triple can then be seen as representing an atomic "fact" or a "claim", and consists of *subject*, *predicate* and *object*. A set of these triples forms an RDF graph.

The prototype research of processing RDF data had a similar start to that of the relational data model. That is, the data are persisted and indexing techniques are utilised on top of it to process it with expressive query languages: SPARQL is the SQL of RDF and triples stores are the relational stores for RDF. In recent years, a number of highly efficient RDF triple stores have been engineered, storing billions of triples with high-speed query processing.

However, in today's application, the assumption of static data may not be applicable, and data items arrive in a continuous, ordered sequence of items. Consider few examples: on social networks, people continuously collaborate, consequently producing data in a continuous manner; sensors, that are ubiquitous devices and crucial for a multitude of applications, continuously produce situational data. Hence, data are always in motion for such applications, construct a dynamic world, and contain an additional attribute of time. Such data are not only produced rapidly, but also continuously – hence forming *data streams*. This highly dynamic and unbounded nature of data streams requires that a new processing paradigm be found: data from a variety of sources are pushed into the system and are processed by persistent and continuous queries, which continuously produce the results with the arrival of new data items.

The apparent characteristics of data streams are especially challenging for both DBMSs and RDF query processors. The reasons are two-fold: first of all, data streams are usually produced at a very high frequency, often in a bursty manner, can pose real-time requirements on processing applications and may only allow them one pass over the data. Second, data streams result in a high volume of data, such that not all of it can be stored and processed. Considering these requirements, traditional DBMSs are simply not suitable to process data streams in a timely fashion. Thus, a new research field, Data Stream Management Systems (DSMSs), was introduced parallel to DBMSs with the following novel requirements:

- The computation is performed in push-based manner or it is data driven. That is, newly arrived data items are continuously pushed into the DSMS to be processed.

- The DSMS's queries are persistent, and continuously processed throughout the lifetime of the streams. The results of these continuous queries also take the form of streams.

- Data streams are considered to be unbounded, thus they cannot be stored in their entirety. Instead, a portion of the recent data items are stored and processed, where the boundaries of recency are defined by the users. These boundaries are generally called *windows*.

- Due to the requirement of a real-time response, DSMSs should employ the main-memory to process the most recent data items within windows.

- New data models and query languages are required to comply with the above mentioned requirements.

Guided by these requirements, a large number of DSMSs have been developed in the last decade or so. These systems employ specialised languages and data structures to optimise response time and improve on scalability requirements.

Following the DSMSs, the Semantic Web community also leaped into this field and named it RDF Stream Processing (RSP). The use of the RDF data model enabled RSP systems to comply with the heterogeneity requirements of today's data sources. Furthermore, the use of ontologies and static background knowledge empowered the RSP systems to extract the contextual knowledge from the dynamic world. However, RSP systems also come with their drawbacks. Having been inspired by DSMSs, they inherit most of their optimisation strategies without explicitly considering the complex nature of RDF. This results in a huge drop in their performance and scalability measures. One can engineer an RDF model on top of a DSMS while employing static data indexing techniques, for example, but it is difficult to optimise it without considering the graph nature of RDF streams and their dynamicity. Nevertheless, these systems pioneered the field of dynamic machine-readable Web, where machines can make decisions in real-time.

As one can imagine, this cannot be the end of the story, and there is a twist in the tail. DSMSs are considered as monitoring applications, where the aim is to process the query operators insipid from SQL such as selection, joins, etc. So the question is *where is the temporal reasoning and how to contemplate the happening of something, i.e., an event?* Complex Event Processing (CEP) was developed to achieve this. Data items are considered as *atomic* events, and the combination of a set of them – which corresponds to the defined temporal patterns – constitute complex/composite events. CEP, however, adheres to the main design principles of DSMSs, but it improves upon them by providing temporal operators. These temporal operators include: sequencing of events, negations, kleene closure, etc. These operators have diverse use cases in many critical applications, such as sensor network, stock market, inventory management, social network analysis, etc. Consider a simple example of events emanating from a nuclear power station. A user is interested in receiving a critical alarm if the temperature of a nuclear power station is greater than a certain threshold value, *followed by* the detection of smoke. Using both of these atomic events, i.e., a rise in temperature, and a presence of smoke, a user can easily infer the complex event of a fire. Note that a DSMS in this case cannot contemplate such a complex event, it can only determine if there is an event of high temperature or take the average of temperature values.

The evolution of CEP systems has started from active databases, where temporal operators were evaluated on persistent data, and currently the popularity of CEP systems has reached the level of DSMSs. Thus, a number of new languages with expressive temporal operators have been proposed, and new techniques have been devised for their efficient implementation. However, such response has not been received from the Semantic Web community, and only handful of solutions have been proposed with restricted functionality. These solutions, usually called Semantic Complex Event Processing (SCEP), have acquired their design directly from standard CEP systems: RDF triples are mapped onto the underlying CEP systems, and subsequently temporal operators are evaluated over them. Therefore, these solutions (i) do not provide all the required/general temporal operators, and (ii) they are not optimised for an RDF data model.

This thesis focuses on both aspects of RDF graph-based stream processing and SCEP. It proposes how to provide an efficient and scalable solution for continuously processing RDF graph streams, and how to build a performance intensive SCEP system with expressive temporal operators. The main directions of our work are: (a) providing

incremental indexing and evaluation for RDF graph streams, (b) providing an expressive SCEP query language, by extending SPARQL, and its efficient implementation. In the following section, we introduce our problem statements and contributions of this thesis.

1.1 Overview and Contributions

This thesis is organised into three main parts: (I) background and existing works on DSMSs, RSP systems, CEP systems and SCEP systems, (II) continuous query processing over RDF graph streams, and (III) SCEP over RDF graph streams. Herein, the main problems discussed in this thesis are summarised under the form of questions.

Table 1.1: Thesis Overview

Part	Research Problem	Chapter
I: Background	What is the **Semantic Web**, and what are its main constituents?	2
	What are the main **lessons** that can be learned from the **existing DSMSs**, and how do **RSP systems** inherit their techniques?	3, 4
	What are the main properties of **CEP query operators and systems**, and what can we learn from **existing SCEP systems**?	5
II: RDF Graph Stream Processing	What are the **limitations of existing solutions**, and how can we implement **incremental indexing and evaluation** techniques for RDF graph streams?	6, 7
III: SCEP	Why do we need a **new SCEP language**, what **operators** should it contain, and how can said SCEP language be **implemented effectively**?	8, 9

Herein, we describe Parts II and III of the thesis and list our contributions.

1.1.1 Part II: Continuous Query Processing over RDF Graph Streams

RDF stream processing (RSP) systems employ RDF triple streams, where each data item within a stream is a single RDF triple ($\langle subject, predicate, object \rangle$). A set of RDF triple streams are matched against an extended form of SPARQL query (query graph) and the matches are propagated as output streams. These systems provide either a black or white box approach. A black box approach directly uses an existing DSMS and a triple store, while a white box approach employs techniques from DSMSs and adapts them for RDF triples. In both cases, a mapping from RDF triple to the underlying tuples is performed and an indexing structure (a B+ tree) is used on top of it.

Problem Statement 1 *What can we learn from existing RSP systems, and how can we provide customised data structures and indexing techniques for RDF graph streams? How can a new solution accommodate both RDF data and streaming requirements in a single framework?*

A direct way of processing RDF graph streams, where each data item is a graph instead of a triple, is to transform static graph solutions for streaming settings. However, these solutions are based on offline indexing and create indices a priori assuming accurate

workload knowledge and data statistics, and plenty of priori slack time to invest in physical design. Furthermore, these solutions are based on an *index-store-query* model, which is not in line with the streaming requirements. RSP systems improve on this by providing online indexing, where the basic concepts of offline indices are transformed into online indices. That is, the system monitors the workload and reorders its operators. However, most of these systems employ statically-optimised structures for such indexing (such as B+ trees). Hence, first these kinds of indexing are not insertion and deletion friendly. Second, in case of variable workloads, the creation of new indices from scratch can considerably outweigh the cost of query processing.

We formalise the problem of continuously processing RDF graph in Chapter 6, which shows that the number of triples within an event has a direct impact on the query cost. Thus, in order to reduce the search space on top of the incremental indexing, we also propose a graph summarisation technique. It employs the structural and selectivity attributes of a query graph and prunes the unnecessary triples from each incoming RDF graph. We list our contributions as follows.

In Chapter 7, we describe our system, called SPECTRA (Chapter 7), that provides answers to the shortcomings of former techniques. It aims at providing an incremental indexing technique that is the by-product of query processing. Thus, it offers considerable advantages over offline and online indexing, which is employed by static RDF and RSP solutions respectively. Furthermore, contrary to existing approaches, we employ an incremental evaluation of triples within a window. That is, with the insertion and eviction of triples, query matches are produced while considering the previously matched results: most of the existing RSP solutions re-evaluate the matched results from scratch. This results in a considerable reduction in response time, while cutting the unnecessary cost imposed by re-evaluation models for each triple insertion and eviction within a defined window.

Contributions:

- *Problem Formulation and Data Model.* We formally introduce the problem and data model for continuously querying RDF graph streams over sliding windows.
- *Graph Summarisation.* We provide a novel graph summarisation technique to prune unwanted triples from each RDF graph within the streams.
- *Incremental Indexing.* We propose an incremental indexing technique for RDF graphs, which is a by-product of query processing and is compatible with the streaming settings.
- *Incremental Evaluation.* Our query evaluation is also based on an incremental model, where the previously matched results are reused.
- *Effectiveness on multiple Datasets.* Our experimental results on both synthetic and real-world datasets show up to an order of magnitude of performance improvements as compared to state-of-the-art systems.

1.1.2 Part III: Semantic Complex Event Processing Over RDF Graph Streams

CEP systems are inspired from active databases, where temporal operators are applied over static datasets. Hence, the data model of most of the existing CEP systems is inspired

by the relational data model. That is, each event contains a relational tuple – with a set of keys and values – associated with a timestamp or time-interval. The simplicity of the data model paves the way to employ expressive temporal operators for CEP query languages, inherit optimisation techniques from DSMSs, and borrow from existing pattern matching techniques. SCEP solutions inspired by CEP systems generate mappings from RDF triples to tuples, and usually employ CEP systems as their underlying execution engines. This first restricts the use of many expressive CEP operators for their query languages due to the complexity of the RDF data model. Second, it hampers their scalability and performance measures: SCEP requires customised optimisations for the RDF data model.

Problem Statement 2 *How can we design a new SCEP query language that covers all the main temporal operators? How can such a language be implemented in an efficient way to provide the scalability and performance requirements?*

Having gained knowledge about efficiently processing RDF graph streams, the question is how to extend it to enable temporal operators over RDF graph-based events. The first answer to this question came with the design of a new query language for SCEP. Existing SCEP query languages do not provide the required capabilities due to the following reasons: (i) they are based on the RDF triple stream data model, (ii) they provide a small subset of temporal operators, (iii) their semantics for matching graph patterns and temporal operators are mixed, making it difficult to extend them for expressive new operators. These are the problems we address in Chapter 8.

Considering the aforementioned shortcomings of existing SCEP query languages, we provide a novel query language called SPAseq (Chapter 8). It extends SPARQL operators with a set of expressive temporal operators. One of the main attributes of SPAseq is that it provides a clear separation between the constructs of graph patterns from SPARQL and temporal operators. Hence, it can easily be extended for the new temporal operators, and its design is not restricted by the underlying executional framework. Moreover, SPAseq data model is based on the RDF graph model, where each event contains a set of RDF triples. In Chapter 8, we provide the syntax and semantics of SPAseq and provide a comparison with existing SCEP languages.

In addition to the syntax and semantics of SPAseq, we also provide its efficient implementation. Since SPAseq provides a clear separation of graph patterns and temporal operators, we employ techniques for graph pattern matching from our system SPECTRA, and a Non-deterministic Finite Automata (NFA) model for the evaluation of temporal operators. Furthermore, we provide various system and operator level optimisation by considering the RDF graph model and lessons learned from existing CEP systems. Such discussion is provided in Chapter 9. We list our contributions as follows:

Contributions:

- *Problem Formulation and Data Model.* We formally introduce the problem and data model for SCEP over RDF graph streams.
- SPAseq *Query Language.* We provide a novel query language for SCEP called SPAseq and provide the syntax and semantics of this query language.
- *Qualitative Analysis.* We provide a qualitative analysis of SPAseq in comparison with existing SCEP languages.

- *Evaluation Framework.* We propose a novel evaluation framework for SPAseq queries, and provide various customised optimisations to evaluate SPAseq query operators.
- *Effectiveness on Multiple Datasets.* Our experimental results on both synthetic and real-world datasets show the effectiveness of these optimisation techniques.

1.2 Research Impact

Parts of the work presented herein have been published in various international workshops and conferences. In the following, we briefly introduce them in the chronological order.

- We presented an ontology design for the RDF graph-based events emanated from a *Smart Grid*, which serves as a precursor to the use cases presented in Chapter 6 and 8: [GLP14] in workshop EnDM@EDBT/ICDT.
- We presented the visionary works regarding the design of a SCEP language, which serves as a precursor to work presented in Chapter 8: [SGL14, GPL14] in workshops Ordring@ISWC and IWWISS.
- We presented our work on continuous graph pattern matching, and data models for the RDF graph streams. It serves as a precursor work to the work presented in Chapter 6 and 7: [GPL16a] in ACM DEBS conference.
- We published an extension of the above paper for our framework SPECTRA (with new algorithms, indexing techniques, and data model). This provides the basis of our work presented in Chapters 6 and 7: [GPL16c] in SSDBM conference.
- Recently, we have also provided a visionary paper to extend the RDF graph streams and SCEP solutions for the distributed environment. This serves as our future work as discussed in Chapter 11: [GPL16b] in workshop GraphQ@EDBT/ICDT.

Beside the above mentioned papers, we have also been involved in other published works which have received much inspiration from this thesis. Highlights include, a paper presented at ACM DEBS on providing a Top-k operator over RDF graph streams [Gil+15]. In this paper, we use techniques for continuously processing RDF graph-based events, and employ a new data structure. It permits the system to capture the top-k elements within a stream with user-defined constraints. Furthermore, another of our papers [Kam+16], published in ACM DEBS, employs incremental indexing on top of relational data streams. This results in providing a scalable solution for the top-k operator over non-linear sliding windows. Moreover, much of the work described in Chapter 8 and 9 will be submitted for review in the Journal of Web Semantics. The complete list of related publications is provided in Appendix B.

Part I

Background and Related Work

I know by my own experience how, from a stranger met by chance, there may come an irresistible appeal which overruns the habitual perspectives just as a gust of wind might tumble down the panels of a stage set – what had seemed near becomes infinitely remote and what seemed distant seems to be close.

— Gabriel Marcel, *On the Ontology of Mystery*

2

Background on Semantic Web Technologies

In this Chapter, we provide a broad overview of the history and key concepts of the Semantic Web. These concepts provide a crucial background to our discussion on the Semantically-enabled Stream Processing and Complex Event Processing. It starts with the evolutionary history of the World Wide Web and then presents the case of the Semantic Web.

Contents

This Chapter is structured as follows: Section 2.1 provides the preliminary history and introduction of the World Wide Web. Section 2.2 provides insight into the Semantic Web. Section 2.3 details the Resource Description Framework (RDF). Section 2.4 describes the syntax and semantics of the SPARQL query language for RDF. Section 2.6 summarises the chapter.

2.1 Introduction

The World Wide Web (WWW) that we all cherish is an ever growing information resource, that is built on the concept of distribution of information with global access. In order to provide the vision of WWW, there are primarily two requirements: machine-readable

structure and a global system of interlinking such structured documents. The first requirement is achieved by providing a Hyper Text Markup Language (HTML) [Nel65], which essentially contains formatted natural language, digital images, etc., while the second goal is realised through globally unique addresses, i.e., Unique Resource Locators (URLs): URL encodes the location from which (and to certain extent, the means by which) a document can be retrieved. By combination of these two features, the structured documents can link/hyperlink to other related documents, embedding the URLs of target documents into the body of text, allowing users to browse between related documents: hence a mesh of interlinked information resources that can be accessed globally.

The WWW project was initiated to share data, of various formats, by physicists at CERN Geneva, Switzerland, and arose from the seminal work by Tim Berners-Lee. It was early 80's, when Berners-Lee started his work on designing a hypertext documentation system called ENQUIRE [BL80]: this laid the ground work, and would foreshadow his later work on the WWW. The aim behind ENQUIRE was to share the complex technical information within the collaborative environment of CERN [BL93]. The ENQUIRE system centred around "cards" as information resources about "nodes", which could refer to a person, a software module, etc., and which could be interlinked using a selection of relations, such as `made`, `includes`, `uses`, `describes` [BL80].

Although, the design of ENQUIRE suited its purpose, it had various limitations [BL93], which include: lack of physical communication layer and file system limited to a local level. Moreover, it required extensive co-ordination to keep information up-to-date. Thus came the concept of WWW to provide a more open and collaborative tool.

> *I wanted [ENQUIRE] to scale so that if two people started to use it independently, and later started to work together, they could start linking together their information without making any other changes. This was the concept of the Web.*
>
> – Berners-Lee [1993]

By late 1990, Berners-Lee had developed initial versions of the technologies underpinning today's Web: the HyperText Markup Language (HTML) used for encoding document formatting and layout, the HyperText Transfer Protocol (HTTP) for client/server communication and transmission of data (HTML) over the Internet, the first Web client software (a "browser" called WorldWideWeb), and a software to run the first Web server.

Later advancements in client and sever-side software brings us to the Web we know today: a highly dynamic, highly flexible platform for hosting, publishing, adapting, submitting, interchanging, curating, editing and communicating various types of content, where many sites boast large corpora of rich user-generated data – typically stored in relational databases – but where the content of different sites is primarily interconnected by generic hyperlinks.

2.2 The Semantic Web

The Web has inarguably been tremendously successful, and begs the question: what's next?

To begin to meaningfully answer this question, one has to look at the shortcomings of the current Web; along these lines, consider querying the question: *Which ten countries have the longest life expectancy and have made advancements in the health and space sectors?* One could hope that: (i) someone has previously performed this task and

published their results, or (ii) a domain-specific site has the data and the functionality required to answer this query directly, or (iii) a domain-specific site has the data available for download in a structured format processable off-line. However, clearly these solutions do not extend to the general case.

Assuming that the above solution does not apply and if a user knows that the data are on the Web, the integration of such data to generate the final answer would require quite a large manual effort. It includes: cross referencing the list of countries' life expectancy numbers, and the statistics about the health and space programs: likely from different data sources. The resulting data may be unstructured or in heterogeneous formats, and the user would like the data in some consistent structured format; such that the user can use a software suitable for such format. Once the data are in a computer-processable format, the user might run into problems with countries names, statistics, as abbreviations may be used.

One can of course imagine variations on the above theme: original search which requires various levels of cross-referencing of various Web documents. Such tasks require: (i) structured data to be made available by the respective sources such that they can be subsequently processed by machine; (ii) some means of resolving the identity of resources involved such that consistent cross-referencing can be performed.

Acknowledging such requirements, Berners-Lee [BL98] proposed the Semantic Web as a variation, or perhaps more realistically, an augmentation of the current Web such that it is more amenable to machine processing, and such that machines can accomplish many of the tasks users must currently perform manually.

> *The Semantic Web is not a separate Web but an extension of the current one, in which information is given well-defined meaning, better enabling computers and people to work in cooperation.*
>
> – Berners-Lee [2001]

2.3 Resource Description Framework

The first major step towards realising the Semantic Web came in early 1999 when the initial Resource Description Framework (RDF) became a W3C Recommendation [Rdf]. RDF provides the basis of an agreed-upon data model for the Semantic Web, and data can be shared/exchanged between RDF-aware agents without loss of meaning [CWL14].

Notably, RDF is (implicitly) based on two major premises.

1. the Open World Assumption (OWA), which assumes that anything not known to be true is unknown, and not necessarily false as would be assumed in closed systems[1];
2. no Unique Name Assumption (UNA), which means that RDF does not assume that a name (in-particular, an URI) signifies something unique: more precisely, the mapping from names to things they identify is not assumed to be injective.

Herein, we give a brief walk-through of the design principles and the features of RDF. We do not cover all features, but rather focus on core concepts that are important for further reading of this document.

[1] Arguably, the SPARQL standard for querying RDF contains features which appear to have a Closed World Assumption (e.g., negation-as-failure is expressible using a combination of `OPT` and `!BOUND` SPARQL clauses) and a Unique Name Assumption (e.g., equals comparisons in `FILTER` expressions). The effects of the Open World Assumption and the lack of a Unique Name Assumption are most overt in Web Ontology Language (OWL) [MH04].

2.3.1 RDF Terms

The elemental constituents of the RDF data model [CWL14] are RDF *terms* that can be used in reference to *resources*: anything with identity. The set of RDF terms is broken down into three disjoint sub-sets: IRIs[2] (or URIs), *literals* and *blank nodes*

1. **IRIs** serve as global (Web-scope) identifiers that can be used to identify any resource. For example, `http://dbpedia.org/resource/Pink_Floyd` is used to identify the music band Pink Floyd in DBpedia[3] [Biz+09] (an online RDF database extracted from Wikipedia content).

2. **Literals** are a set of lexical values denoted with inverted commas in Turtle, N3 or other RDF formats[4]. Literals can be of two different types: *plain literals*, which form a set of plain strings, such as "`Hello World`", potentially with an associated language tag, such as "`Hello World`"`@en`; *typed literal*, which comprise of a lexical string and a datatype such as "`2`"`^^xsd:int`. Datatypes are identified by IRIs (such as `xsd:int`), where RDF borrows many of the datatypes defined for XML Schema that cover numerics, booleans, dates, times, and so forth.

3. **Blank Nodes** are defined as existential variables used to denote the existence of some resources without having to explicitly reference it using an IRI or literal. In practice, blank nodes serve as locally-scoped identifiers for resources that are not otherwise named. Blank nodes cannot be referenced outside of their originating scope (e.g., an RDF document). The labels for blank nodes are thus only significant within a local scope. Intuitively, much like variables in queries, the blank nodes of an RDF document can be relabelled (bijectively) without affecting the interpretation of the document [Hog+14]. In Turtle (verbose style), blank nodes can be referenced explicitly with an underscore prefix `_:bnode1`, or can be referenced implicitly (without using a label) in a variety of other manners.

We can now provide a formal notation for referring to different sets of RDF terms:

Definition 2.1: RDF terms

The set of RDF terms is the union of three pair-wise disjoint sets: the set of all IRIs ($\mathcal{I}$), the set of all literals ($\mathcal{L}$) and the set of all blank nodes ($\mathcal{B}$). The set of all literals can be further decomposed into the union of two disjoint sets: the set of plain literals ($\mathcal{L}_p$) and the set of typed literals ($\mathcal{L}_t$).

In the absence of Unique Name Assumption, as described in the RDF standard, two RDF terms can (and often do) refer to the same referent. Since RDF is intended to be used as a common data model for the Web, it is likely that two different publishers may use different terms to refer to the same thing or entity[5].

[2] Following the RDF 1.1 vocabulary, in the rest of the documents we use the term IRI instead of URI

[3] DBpedia: `http://wiki.dbpedia.org/`

[4] By RDF format, we mean the way RDF data are serialised in order to be stored or transferred between machines. In general, there are three main ways of formatting RDF: RDF/XML, as evident from its name, it uses XML formatting; Turtle (Terse RDF Triple Language) is an RDF-specific subset of Tim Berners-Lee's Notation3 language; and N-triples, which is a simplified version of Turtle, where each triple appears on one line, separated by a dot.

[5] Herein, by thing or entity, we mean the concept defined within a domain ontology. For instance, a

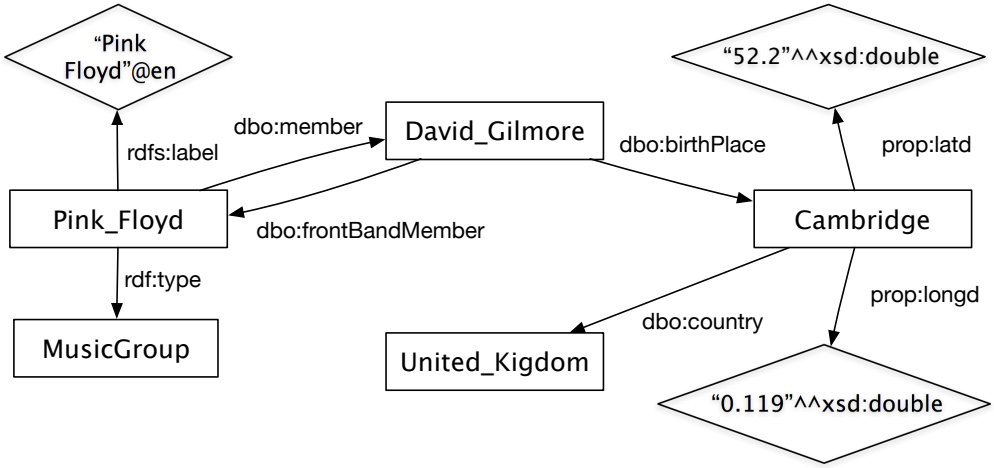

Figure 2.1: An Example of an RDF Graph

2.3.2 RDF Triples and Graphs

RDF triples, that are based on RDF terms, are used to make statements about the things. The notion of RDF triple constitutes the foundation of the Semantic Web's core data model. As its name suggests, an RDF triple is a 3-tuple of RDF terms. The first element of the tuple is called the *subject*, the second element the *predicate*, and the third element the *object*. An RDF triple can be seen as representing an atomic "fact" or a "claim". Importantly, RDF triples have fixed arity (length of three) with fixed slots $\langle subject, predicate, object \rangle$, constituting a generic common framework that enables interoperability. As aforementioned, it can be used to designate classes to resources. Figure 2.1 shows an exemplary RDF graph (resources are shown within a rectangle, while literal-valued attributes in a diamond), where

(`Pink_Floyd, rdf:type, MusicBand`)
is used to define a resource, and to define a literal-valued attribute for a resource:
(`Cambridge, prop:longd, "52.2"^^xsd:double`)
Formally, an RDF triple can be defined as follows [CWL14]:

Definition 2.2: RDF Triple

*An **RDF triple** t is defined as a triple $t = \langle s, p, o \rangle$, where $s \in \mathcal{I} \cup \mathcal{B}$ is called the subject, $p \in \mathcal{I}$ is called the predicate and $o \in \mathcal{I} \cup \mathcal{B} \cup \mathcal{L}$ is called the object.*

Definition 2.3: RDF Graph

*An **RDF graph/RDF dataset** $G_D \subset (\mathcal{I} \cup \mathcal{B}) \times \mathcal{I} \times (\mathcal{I} \cup \mathcal{B} \cup \mathcal{L})$ is a finite set of RDF triples $\{(s_1, p_1, o_1), \ldots, (s_n, p_n, o_n)\}$.*

Since RDF graphs are defined in terms of sets, it follows that the ordering of RDF triples in an RDF graph is entirely arbitrary and that RDF graphs do not allow for duplicate triples. It is a common practice to conceptualise an RDF graph as a directed

Jaguar, which is a thing, can be a car or an animal depending to how it is defined within the domain ontology.

labelled graph, where subjects and objects are drawn as labelled vertices and predicates are drawn as directed, labelled edges. Although some authors have suggested alternative representations such as bipartite graphs [HG04], directed labelled graphs remain an intuitive and popular conceptualisation of RDF data. As such, RDF is often referred to as being graph-structured data where each $\langle s, p, o \rangle$ triple can be seen as an edge $s \xrightarrow{p} o$.

The graph-structured nature of the RDF data model lends itself to a flexible integration of datasets. Edges in the graph use globally-scoped IRI identifiers. When vertices are identified with IRIs, they can be referenced externally and connected to other vertices. However, due to the presence of blank nodes and the fact that the predicates can be used as subjects and objects, the RDF data model is not completely isomorphic to the notion of directed-labelled graphs. Such constraints require customised solutions to process RDF data (as described in Section 7).

2.3.3 Linked Data

Publishing RDF data on the Web facilitate enhanced methods to obtain knowledge and enable the construction of new types of front-end applications. However, early efforts produced a large amount of "data soils" often in the shape of potentially huge RDF documents called RDF data dumps. Although such dumps have their own inherent value and are published in an interoperable data-model through RDF, they rarely interlink with remote data and they are published using different conventions (e.g., in a Web folder, using different archiving methods, etc.). Thus, this makes them difficult to be discovered automatically. Effectively, such dumps are isolated islands of data that are available for download, or offer a SPARQL access point.

Linked Data [BL06] is a set of best practices for publishing and interconnecting structured data on the Web, i.e., RDFised data dumps. Linked Data provides explicit links between data from diverse sources, where IRIs are the means for connecting and referring between various entities from domains such as social networks, organisational structures, government data, statistical data and many others. The ultimate benefit of following the Linked Data paradigm is the increased machine-readability of published and interconnected data. In July 2006, Berners-Lee published the initial *W3C Design Issues* document [BL06] outlining Linked Data principles, rationale and some examples. Herein, we summarise them for completeness.

- **Assign IRIs to entities.** Published entities should have their IRIs which map over the HTTP protocol to their RDF representation. For example, each sensor should have a unique IRI, which links to its information in RDF.
- **Set RDF links to other entities on the Web.** Published entities should be linked with other entities on the Web. For example, when providing the list of sensor functionalities, they should link to the IRIs which describe the details of them in RDF.
- **Provide metadata about published data.** Published data should be described by the means of metadata to increase their usefulness for the data consumers. Data should contain information on their creator, creation date and creation methods. Publishers should also provide alternative means for accessing their data.

The central novelty of Linked Data when compared with traditional Semantic Web publishing was the emphasis on using de-referenceable IRIs to name things in RDF. Thus,

the data published under the Linked Data Principles can be searched through the resource identified by a particular IRI, i.e., through HTTP using content-negotiation methods.

Linked Data has attracted considerable interest from both academic and industrial personals in the last few years. Early adopters included mainly academic researchers and developers, while some of the most prominent examples of the organisations publishing RDF as Link Data include: BBC music data[6], British government data[7] or Library of Congress data[8]. At the same time, an increasing number of public vocabularies (ontologies) and their inter-connectedness are created, which forms a Linked Data Cloud[9].

2.4 The SPARQL Query Language

The SPARQL[10] Query Language is the standardised language for querying RDF data, as well as a protocol by which SPARQL queries can be invoked and their results returned over the Web [PS08]. The original SPARQL specification became a W3C Recommendation in 2008 [CFT08], while in 2013, SPARQL 1.1 – an extension of the original SPARQL standard – also received the W3C Recommendation and later became a standard [SH13]. Herein, we focus primarily on the features of the original SPARQL standard that are helpful for the understanding of discussion in this document.

SPARQL is built directly on top of the RDF data model, and is orthogonal to the RDF schema[11] and OWL languages [12]. That is, it is not intended to offer any reasoning capabilities, instead it provides the graph pattern matching support for RDF graphs. It is similar in respect to the Structured Query Language (SQL) used for querying relational databases.

In general, a SPARQL query consists of five main parts as described below.

1. **Prefix Declarations** allow for defining IRI prefixes that can be used for shortcuts later in the query.

2. **Dataset Clause** allows for specifying a closed partition of the indexed dataset over which the query should be executed.

3. **Result Clause** allows for specifying what type of SPARQL query is being executed, and (if applicable) what results should be returned.

4. **Query Clause** allows for specifying the query patterns (triple patterns as described in Definition 2.4) that are matched against the data and used to generate the variable bindings of the defined variables in the query.

[6]British Broadcasting Company: `http://www.bbc.co.uk`, last accessed: June, 2016.
[7]British government: `http://data.gov.uk`, last accessed: June., 2015.
[8]Library of Congress: `http://id.loc.gov`, last accessed: June., 2016.
[9]LoD cloud: `http://lod-cloud.net`, last accessed: June, 2016.
[10]W3C SPARQL: `https://www.w3.org/TR/sparql11-query/`
[11]RDF schema (RDFS) extends the RDF vocabularies, i.e., a set of built-in vocabulary terms under a core RDF *namespace* (a common IRI prefix schema) that standardises popular RDF patterns (e.g., `rdf:type`, `rdf:Property`). Thus, attaching semantics to the user-defined classes and properties. The extension consists of four key terms [MPG09], which allows specification of well-defined relationships and properties between classes and properties. It includes `rdfs:subClassOf`, `rdfs:subPropertyOf`, `rdfs:domain`, and `rdfs:range`.
[12]OWL is a Web Ontology Language [MH04] that extends RDFS with more expressive semantics and enables rich entailment regimes. The details of OWL are not important for our purpose and are thus considered out-of-scope.

5. **Solution Modifiers** allow for ordering, slicing and paginating the results.

Example 1 *Query 2.1 illustrates a simple SPARQL query containing each of the above mentioned five parts (comment lines are prefixed with '#'). This SPARQL query first defines prefixes that can be later re-used as shortcuts to the resources. Next the* `#DATASET CLAUSE` *selects partitions of the dataset over which the query should be run: in this case, an RDF documents from DBpedia containing the information about bands. Thereafter, the* `#RESULT CLAUSE` *states what kind of results should be returned for the query. A* `DISTINCT` *keyword is used to get the unique set of pairs of matched RDF terms matching the* `?bandname` *and* `?genre` *variables respectively. Next the* `#QUERY CLAUSE` *states the patterns that the query should match against, i.e., the set of triple patterns defined. Finally, the* `#SOLUTION MODIFIER` *section allows for putting the limit on the number of results returned, to order results, or to paginate results.*

```
# PREFIX DECLARATION
PREFIX db: <http://dbpedia.org/resource/>
PREFIX dbo: <http://dbpedia.org/ontology/>
# DATASET CLAUSE
FROM <http://dbpedia.org/data/example.n3>
# RESULT CLAUSE
SELECT DISTINCT ?bandname ?genre
# QUERY CLAUSE
 WHERE {
    ?band dbo:name ?bandname.
    ?band dbo:genre ?genre.
                  }
#SOLUTION MODIFIER
 LIMIT 2
```

Query 2.1: Simple SPARQL query

The execution of a SPARQL `SELECT` query is not itself a graph. Similar to SQL, it is a set of rows of mappings of selected variables. Thus, in order to extract/create a graph from the resulted SPARQL variable binding, the `CONSTRUCT` clause can be used at the result clause. Furthermore, the `ASK` construct at result clause returns a boolean value indicating whether or not there was a match in the data for the query clause.

2.4.1 Semantics of the SPARQL Query Evaluation

Herein, we describe the semantics of evaluating a SPARQL query on the same line as described in [PAG09a]. For the sake of brevity, we employ the set-based semantics of SPARQL. The notion of a triple pattern is defined as follows.

Definition 2.4: Triple Pattern

Let $\mathcal{V}$ be a set of query variables disjoint from $\mathcal{B} \cup \mathcal{L} \cup \mathcal{I}$, then a ***triple pattern*** *$tp \in (\mathcal{B} \cup \mathcal{I}) \times (\mathcal{V} \cup \mathcal{I} \cup \mathcal{V}) \times (\mathcal{L} \cup \mathcal{I} \cup \mathcal{B} \cup \mathcal{V})$ is a triple where query variables are allowed at subject, predicate and object levels.*

Definition 2.5: SPARQL Graph Pattern

The SPARQL graph patterns (or a basic graph pattern) is defined recursively as follows.

1. *A triple pattern tp is a graph pattern.*
2. *If P_1 and P_2 are graph patterns, then the expression (P_1 `AND` P_2), (P_1 `OPT` P_2), (P_1 `UNION` P_2)* are graph patterns.
3. If P is a graph pattern and R is a SPARQL build-in condition, then the expression (P `FILTER` R) is a graph pattern.

A SPARQL built-in condition is a boolean combination of terms constructed by using equality along elements in $\mathcal{I} \cup \mathcal{L} \cup \mathcal{V}$, and the unary predicate *bound* over variables, i.e., it returns *true* if the variable is bound to a value. Formally it can be defined as follows.

1. if $x, y \in \mathcal{V}$ and $c \in (\mathcal{I} \cup \mathcal{L})$, then $bound(x)$, $x = c$ and $x = y$ are built-in conditions;
2. if R_1 and R_2 are built-in conditions, then $\neg(R_1)$, $R_1 \vee R_2$, and $R_1 \wedge R_2$ are build-in conditions.

Example 2 *Consider the following SPARQL graph pattern with the filter expression, which retrieves the bands with the name "Pink Floyd"*

((?band, hasName, ?name) `AND` (?band, hasGenre, ?genre) `FILTER` (?name = "Pink Floyd"))

The semantics of SPARQL is basically defined using the concepts of mappings, which express variable-to-RDF bindings during query evaluation.

Definition 2.6: Triple Pattern Mapping

A mapping (μ) is a partial function $\mu : \mathcal{V} \rightarrow \mathcal{B} \cup \mathcal{L} \cup \mathcal{I}$ from a subset of variables $\mathcal{V}$ to RDF terms. The domain of a mapping μ, $dom(\mu)$ is the subset of V for which μ is satisfied. We say that two mappings μ_1, μ_2 are compatible, written as $\mu_1 \sim \mu_2$, if they agree on all the shared variables. That is, if $\mu_1(x) = \mu_2(x)$ for all $x \in dom(\mu_1) \cap dom(\mu_2)$.

Let $vars$ be a function to extract the mappings from the triple patterns and the filter conditions, then $vars(tp)$ captures all the variables in a triple pattern tp. Furthermore, the function $\mu(tp)$ is used to obtains the triple pattern by replacing all the variables, i.e., $x \in dom(\mu) \cap vars(tp)$ in tp by $\mu(x)$.

Example 3 *Consider the three mappings $\mu_1 := \{x \mapsto k\}$, $\mu_2 := \{x \mapsto k, y \mapsto k2\}$, and $\mu_3 := \{x \mapsto k2, z \mapsto k3\}$, where $x, y, z \in \mathcal{V}$ and k, k2, k3 $\in (\mathcal{B} \cup \mathcal{L} \cup \mathcal{I})$. Then we can see that $dom(\mu_1) = \{x\}$, $dom(\mu_2) = \{x, y\}$, and $dom(\mu_3) = \{x, z\}$. Next we can also determine that $\mu_1 \sim \mu_2$, however $\mu_1 \nsim \mu_3$ and $\mu_2 \nsim \mu_3$. Given triple pattern tp := (f, x, y) we have $vars(tp) = \{x, y\}$ and $\mu_2(tp) = (f, k, k2)$.*

The semantics of the filter built-in condition are also defined using the concept of mapping. A mapping μ satisfies the filter conditions $bound(x)$ if the variable x is contained

in $dom(\mu)$; the filter conditions $x = c$, $x = y$ and $c = d$ are equality checks that compare the value of $\mu(x)$ with c, $\mu(x)$ with $\mu(y)$, and c with d respectively. These checks fail whenever one of the variables is not bound in μ. Furthermore, the boolean conditions on these variables are defined in the usual way of boolean comparison, and a mapping μ that satisfies the filter condition R is written as $\mu \models R$. The complete semantics of the SPARQL graph patterns and their operators are described using the set of mappings (Ω) [PAG09a].

Definition 2.7: SPARQL Set Algebra

Let Ω, Ω_1, and Ω_2 be three sets of mappings, R be a filter condition, $v \subset \mathcal{V}$ be a finite set of variables. We define the algebraic operations join ($\bowtie$), union ($\cup$), minus ($\setminus$), left outer join ($\leftouterjoin$), projection (π), and selection (γ) as follows:

- $\Omega_1 \bowtie \Omega_2 := \{\mu_1 \cup \mu_2 \mid \mu_1 \in \Omega_1 \wedge \mu_2 \in \Omega_2 \wedge \mu_1 \sim \mu_2\}$
- $\Omega_1 \cup \Omega_2 := \{\mu \mid \mu \in \Omega_1 \vee \mu \in \Omega_2\}$
- $\Omega_1 \setminus \Omega_2 := \{\mu_1 \in \Omega_1 \mid \forall \mu_2 \in \Omega_2 \wedge \mu_1 \nsim \mu_2\}$
- $\Omega_1 \leftouterjoin \Omega_2 := (\Omega_1 \bowtie \Omega_2) \cup (\Omega_1 \setminus \Omega_2)$
- $\pi_v(\Omega) := \{\mu_1 \mid \exists \mu_2 : \mu_1 \cup \mu_2 \in \Omega \wedge dom(\mu_1) \subseteq v \wedge dom(\mu_2) \cap v = \varnothing\}$
- $\gamma_R(\Omega) := \{\mu \in \Omega \mid \mu \models R\}$

In order to define the evaluation of SPARQL query patterns, as described in Definition 2.5, we follow the compositional semantics from [PAG09a] and define a function $[\![\cdot]\!]_D$, where D is an RDF document, which translates query patterns to the SPARQL set algebra.

Definition 2.8: Evaluation of Graph Patterns

Let D be an RDF document, tp a triple pattern, P, P_1 and P_2 SPARQL graph patterns, R a filter condition, and $v \subset \mathcal{V}$ a set of variables. The semantics of SPARQL graph patterns are defined as follows:

- $[\![tp]\!]_D := \{\mu \mid dom(\mu) = vars(tp) \quad \wedge \quad \mu(tp) \in D\}$
- $[\![P_1 \texttt{ AND } P_2]\!]_D := [\![P_1]\!]_D \bowtie [\![P_2]\!]_D$
- $[\![P_1 \texttt{ OPT } P_2]\!]_D := [\![P_1]\!]_D \leftouterjoin [\![P_2]\!]_D$
- $[\![P_1 \texttt{ UNION } P_2]\!]_D := [\![P_1]\!]_D \cup [\![P_2]\!]_D$
- $[\![P_1 \texttt{ FILTER } R]\!]_D := \gamma_R([\![P]\!]_D)$
- $[\![\texttt{ SELECT}_v(P)]\!]_D := \pi_v([\![P]\!]_D)$

Example 4 *Consider the following SPARQL* `SELECT` query patterns
$G_Q :=$ `SELECT` *?b, ?n, ?g (((?b, hasName, ?n)* `AND` *(?b, hasGenre, ?g))*
`FILTER` *(?g =“Rock”))*

The above mentioned query retrieves all the bands (*?b*) and their names (*?n*) that are from the genre (*?g*) *"Rock"*. Then giving the following RDF dataset D,

$D := \{$*(B1, hasName, "Pink Floyd"), (B1, hasGenre, "Rock")* $\}$

It can be easily verified that,

$[\![G_Q]\!]_D = \{\{?b \mapsto B1\}, \{?n \mapsto$ "*Pink Floyd*"$\}, \{?g \mapsto$ "*Rock*"$\}\}$

2.4.2 Complexity of SPARQL

In this section, we provide the preliminary discussion about the complexity measures of evaluating SPARQL queries. This allows to establish a deep understanding of the SPARQL query operators, their complexity, and their interactions. Furthermore, the complexity analysis of the SPARQL queries may be of immediate practical interest when processing SPARQL queries continuously over RDF graph streams. Such discussion is provided in Chapter 6.

Following the same principle used in [PAG09a], we use the decision version of Evaluation problem as a yardstick to explain the complexity measures of SPARQL query operators. That is, given a mapping μ, an RDF dataset D, and a SPARQL expression or query G_Q as input, we are interested in the complexity of deciding whether μ is contained in the result of evaluating G_Q on D. In order to describe the complexity measures of different segments of SPARQL, we first introduce various shorts: $A :=$ `AND`, $F :=$ `FILTER`, $O :=$ `OPT`, and $U :=$ `UNION`. For notational convenience, we denote the class of SPARQL expressions that can be constructed using a set of operators, and the triple patterns, by concatenating the respective operator shortcuts. For instance, the class AU comprises all SPARQL expressions that can be constructed using only operators `AND`, `UNION`, and triple patterns.

In the subsequent complexity study, we follow the approach from [PAG09a] and take the complexity of the Evaluation problem as a reference:

> *Given a mapping μ, an RDF document D, and a SPARQL expression or a SPARQL query G_Q as input: is $\mu \in [\![G_Q]\!]_D$?*

The following theorem summarises all previous results on the combined complexity of SPARQL fragments established in [PAG09a], and rephrased according to the notations defined above. We refer the interested reader to the original work for the proofs of these results, and the introductory discussion about the complexity classes. That is, PTime $\subseteq$ NP $\subseteq$ PSpace.

Theorem 2.1

(see [PAG09a]) The Evaluation *problem is*

1. *in* PTime *for class AF (membership in* PTime *for A and F follows directly),*
2. NP*-complete for class AFU, and*
3. PSpace*-complete for classes AOU and SP.*

Theorem 2.1 shows the basic complexity classes the SPARQL operators belong to. However, in order to get the complexity of evaluating SPARQL graph pattern (P), according to the dataset size, we present the following theorem.

Theorem 2.2

(see [PAG09a]) The Evaluation *of AFU can be solved in* $\mathcal{O}(|P| \cdot |D|)$

Theorem 2.2 shows that the size of the RDF dataset D has a linear impact on the performance of the SPARQL queries. Thus, a system can gain fair amount of performance by reducing the search space or locating the smallest possible portion of the dataset that requires processing to match query patterns. This insight is utilised by our system to prune the irrelevant triples from each RDF graph-based event; such discussion is detailed in Chapter 7.

2.5 Common Symbols

We give the most common symbols and their short descriptions in Table 2.1. Additional symbols necessary to explain the proposed methods and algorithms are provided in the corresponding chapters.

Table 2.1: Common Symbols and Definitions

Symbols	Description
G_D	RDF graph
$\mathcal{I}$	Set of IRIs
$\mathcal{L}$	Set of Literals
$\mathcal{B}$	Set of Blank nodes
$\langle s, p, o \rangle$	RDF triple t
G_Q	SPARQL query
tp	Triple pattern
P	SPARQL graph pattern
R	SPARQL filter expression
μ	Triple pattern mapping
Ω	Set of mappings
$\sim$	Compatibility between mappings
$\bowtie$	Join between mappings
$\setminus$	Set minus
π	Projection
γ	Selection
$\leftouterjoin$	Left-outer join between mappings
$[\![\cdot]\!]$	Evaluation function

2.6 Summary

In this chapter, we discussed the evolution of WWW from a source of sharing structured documents to sharing machine-readable information along-with the semantics. In order for machines to process the content of documents automatically–for whatever purpose–they primarily require two things: machine-readable structure and semantics. The Semantic

Web provides these attributes with the RDF data model and ontologies. SPARQL is a standard query language for processing RDF datasets, where its core components fall under the PSPACE complexity class. In Chapter 6 and 7 we use this introductory discussion to continuously process SPARQL graph patterns over RDF graph streams, and in Chapter 8 we extend SPARQL query language to support operators for SCEP over RDF graph streams.

Real life is, to most men, a long second-best, a perpetual compromise between the ideal and the possible; but the world of pure reason knows no compromise, no practical limitations, no barrier to the creative activity embodying in splendid edifices the passionate aspiration after the perfect from which all great work springs.

— Bertrand Russell

3 Data Stream Processing

This chapter introduces the concept of data stream processing by describing the details of Data Stream Management Systems (DSMSs). We provide the discussion about the core concepts of DSMSs, an analysis of existing DSMSs, their query languages, operators and execution models. Based on this discussion, in Chapter 4, we present the semantically-enabled stream processing and show how techniques from DSMSs are tailed for such purpose.

Contents

This Chapter is structured as follows: Section 3.1 presents an introductory overview of DSMSs. Section 3.2 discusses the general data models of DSMS. Section 3.3 describes the query languages and executional semantics for DSMSs. Section 3.5 compares various DSMSs according to their provided functionalities. Section 3.6 briefly describes various optimisation techniques exploited by DSMSs. Section 3.7 concludes the chapter with a summary and a discussion.

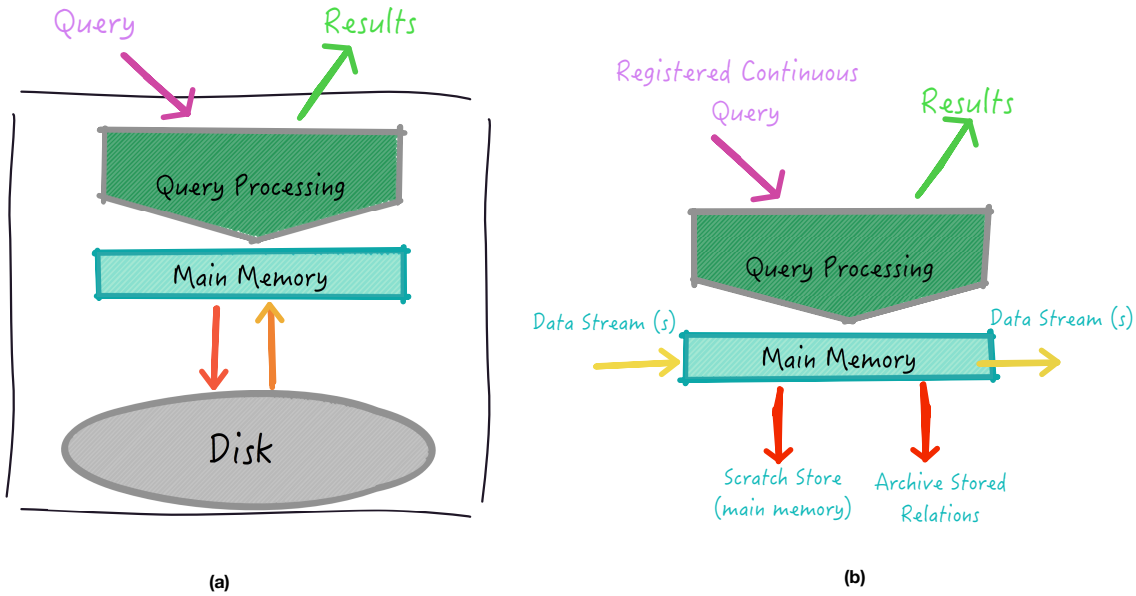

Figure 3.1: (a) Traditional DBMS vs (b) DSMS

3.1 Data Stream Management System

Database Management Systems (DBMSs) have become ubiquitous as a fundamental tool for managing information. DBMSs are used to store, manipulate and retrieve persistent data from a database; thus the dynamicity of data is not considered as an integral part of the system's design phase: it is assumed that data are static, unless explicitly modified or deleted by a user or application, and the queries when issued reflect the current state of the data. Data Stream Management Systems (DSMSs), however, are based on the orthogonal assumption that new data are generated continuously and queries are processed continuously. It makes them feasible for real-time monitoring of emerging applications such as sensor networks, social networks, financial trading, etc. As data are generated continuously as streams, it is infeasible to store the streams in their entirety. Therefore, generally a window of recently arrived data is maintained and the registered queries update their answers over time. The window size can be defined as a fixed number of data items, also called snapshots [KS09a], or a fixed time interval. In the latter case, a *slide* parameter is also introduced to determine the granularity at which the content of the window changes [KS09a]. Figure 3.1 shows the query processing mechanism for a traditional DBMS and DSMS. As the goals of the DSMSs are orthogonal to that of DBMSs, a set of rules has been introduced in the preliminary work [ScZ05]. These eight basic rules lay out the requirements and constraints for the design of a DSMS, and are summarised as follows.

Rule 1 *Keep the data moving.* The processing model should be active and data driven. That is, the data items should be processed in an online manner without incurring the expensive cost of storage before initiating the analysis.

Rule 2 *Enabling streaming semantics for DSMS query languages.* Historically, streaming applications build a new layer on top of existing DBMSs while utilising their query languages such as SQL. However, in order to address the unique requirements of stream processing, DSMS should provide new query languages with extended

streaming operators, such that their executional semantics can easily be understood independently from the runtime conditions.

Rule 3 *Handle stream imperfections.* Contrary to the DBMSs, the data are never stored for the DSMSs and thus the design of the system should consider contingency plans as a result of data arrival delays, absence/lost of data, and out-of-the-order data items. These issues arise frequently in real-world systems such as sensor networks.

Rule 4 *Generate predictable results.* The property of determinism for the processed results should be ensured. That is, the system must compute the equivalent results for two equivalent streams.

Rule 5 *Integration of stateful operators.* The past can revile much important information about the present and future. Thus, the integration of stateful operators, such as sequence, aggregates, provides interesting features; and the stored states should be carefully managed by the system. Furthermore, the management of stored states also enables an effective fault-tolerance for the system in case of abrupt failures.

Rule 6 *Guarantee data safety and availability.* Real-time processing systems are often considered as critical: failure or loss of information can be too costly. Therefore, high availability and resistance to failures are two important properties to be considered for the design of DSMS.

Rule 7 *Automatic partition and scale.* Due to the unbounded nature of streams, the system should be able to transparently distribute its workload among multiple machines and processors, hence improving its scalability.

Rule 8 *Process and respond in real-time.* The foundation of an efficient DSMS is that it can process data attributes with high volume and velocity. Thus, conceiving low latency to enable a real-time response is the core of the DSMS. These attributes can be achieved by employing optimised query-plans for streaming data and by minimising processing overheads.

Rules 1, 2, 4 and 8 are the most commonly observed in most of the exiting DSMSs [Ara+04, Cha+03, YG02, Cra+02, ABW06, NCT08, Aba+03]. Rule 5 gives rise to the definition of Complex Event Processing (CEP) [CM12], where stateful operators are utilised to enable temporal pattern matching. The combination of rules 3 and 6 led to the techniques of load shedding [BDM07], where some data items are discarded to fulfil quality of service (QoS) and real-time constraints.

For brevity, a discussion of the following topics has been omitted: these are not directly related to the topic of this thesis, i.e., semantically-enabled stream processing, and are not used or can be directly extended for such work.

1. Application specific DSMS issues and solutions, such as stream processing for sensor networks. See [SG07] for a recent survey.

2. Distributed stream processing. See for example,

   (a) Open-source system such as Apache Storm[1], Apache Flink[2], Apache Kafka[3], etc.

[1] Apache Storm: `http://storm.apache.org/`, last accessed: June, 2016.
[2] Apache Flink: `https://flink.apache.org/`, last accessed: June, 2016.
[3] Apache Kafka: `http://kafka.apache.org/`, last accessed: June, 2016.

(b) Recent work on Distributed DSMS [Neu+10, Hei+14].

3. Approximate and out-of-order stream processing techniques, see for example the following representative papers [DGR03, Let+10, Li+08, Li+07].

3.2 Data Models for the DSMSs

3.2.1 Data Streams

A data stream is an append-only sequence of timestamped data items that arrive in some order [GM06]. Since items may arrive in bursts, a stream may instead be modelled as a sequence of sets (or bags) of elements [Tuc+03], with each set storing elements that have arrived during the same unit of time[4]. In relation-based stream models (e.g., STREAM [ABW06]), individual items take the form of relational tuples such that all tuples arriving on the same stream have the same schema. The data stream items may contain explicit source-assigned timestamps or implicit timestamps assigned by the DSMS upon arrival. In either case, the timestamp attribute may or may not be part of the stream schema, and therefore may or may not be visible to the users.

Definition 3.1: Relational Data Stream

A relational data stream (S_d) is a countable infinite set of data items ($d \in S_d$), where each data item is a pair (v, τ), $v \in \Re$ is a relational tuple, and $\tau \in \mathbb{N}^+$ is a timestamp and a member of a totally ordered set of timestamps.

3.2.2 Temporal Models of Data Streams

The arrival order of data items, if application time is used, mainly determines the type of the model: data items within a stream may arrive out-of-order and the defined model has to take into account revision tuples, which are understood to replace previously reported (presumably erroneous) data. Some possible models depending on the arrival order of data items are described as follows [Gil+01].

1. *Unordered cash register.* This is most general model, where individual data items from multiple data streams arrive in no particular order and usually application timestamps are utilised to process streams.

2. *Ordered cash register.* In this model, individual data items within streams are not pre-processed to ensure the order. Instead, they arrive in some known order, e.g., timestamp order, where system time is usually used to assign timestamps to each data item.

3. *Unordered aggregate.* This is the aggregate case of unordered cash register, where individual data items from the same stream are pre-processed in no particular order.

4. *Ordered aggregate.* This is the aggregate case for the ordered cash register, where individual items from the same domain are pre-processed in some known order.

[4] no order is specified among data items that have arrived at the same time

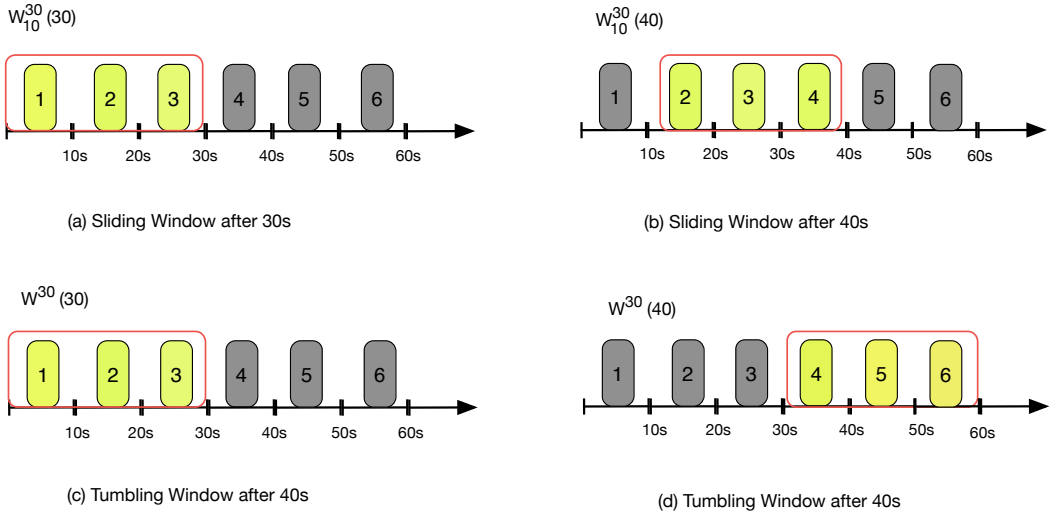

Figure 3.2: (a,b) Sliding Window , (c,d) Tumbling Window, where W_x^w, x is the slide, and w is the size of the window.

3.2.3 Windows

As discussed earlier, windows are a central concept in DSMS because an application cannot store an infinite stream in its entirety: windows are operators that only select a part of the stream according to fixed parameters, such as the size of the window. Hence, they provide an approximation of the stream, but are at the same time implementing the desired query semantics [Bab+02]. Herein, we first describe the most commonly used time-based *sliding window* [KS09a] and *tumbling window* for the DSMSs.

Definition 3.2: Sliding Window

*At a time τ, a **sliding window** $W_x^w(\tau)$ of size w and slide x, where $w, x \in \mathbb{N}^+$, begins at τ_b and ends at τ_e. Such that:*

$$W_x^w(\tau) = \{v | (v, \tau') \in S_d \wedge \tau_b \leq \tau' \leq \tau_e\}, \text{ where}$$
$$\tau_b = \left\lfloor \frac{\tau - w}{x} \right\rfloor \cdot x, \text{ and } \tau_e = \tau_b + w$$

The sliding window for $w = x$ degenerates to a *tumbling window*, where all the data items within a window expire at the same time.

Definition 3.3: Tumbling Window

*At a time τ, a **tumbling window** $W^w(\tau)$ of size $w \in \mathbb{N}^+$ begins at τ_b and ends at τ_e. Such that:*

$$W^w(\tau) = \{v | (v, \tau') \in S_d \wedge \tau_b \leq \tau' \leq \tau_e\}, \text{ where}$$
$$\tau_b = \left\lfloor \frac{\tau - w}{w} \right\rfloor \cdot w, \text{ and } \tau_e = \tau_b + w$$

Time-based windows can easily be extended for a *tuple-based* window, where the size

of the window determines how many explicit tuples are allowed within the boundaries of the window. Note that, other flavours of windows including value-based window, jumping windows and non-linear windows are not discussed here, as they are system-specific windows and are not generalised in the literature. Figure 3.2 shows a sliding and tumbling window, where multiple sliding windows can share the same data items, while all the data items expire at the same time in tumbling windows.

3.3 Query Languages for DSMSs

Queries over continuous data streams have much in common with queries in a traditional database management system. However, the important distinctions peculiar to the data stream model is between *one-time queries* and *continuous queries* [GO03, Bab+02]. One-time queries (a class that includes traditional DBMS queries) are evaluated once over a point-in-time snapshot of the data set, with the answer returned to the user. A continuous query for DSMS is issued once and remains active throughout the lifespan of the streams. The answer to a continuous query is constructed progressively as new input data items arrive: as soon as a data item arrives in the input stream, the DSMS is expected to decide, in real-time or quasi real-time, which additional results belong to the query answer and promptly append them to the output stream.

This is an incremental computation model, where no output can be taken back; therefore, the DSMS might have to delay returning an output tuple until it is sure that the tuple belongs to the final output: a certainty that for queries is only reached after the DSMS has seen the whole input. The queries showing this behaviour, and operators causing it, are called *blocking operators*, and have been characterised in [Bab+02] as follows: *a blocking query operator does not append anything in the output stream until it has seen the entire input.* Clearly, blocking query operators are incompatible with the computation model of DSMS and should be disallowed, whereas all non-blocking queries should instead be allowed. However, many queries and operators, including essential ones such as union, fall in-between and are only partially blocking.

For the sake of brevity, we do not include the discussion of approximate queries [Cor11, GK02], and ad-hoc queries. Various approximation algorithms are used to process the approximate answers to continuous queries, while considering that high-quality approximate answers can be acceptable in lieu of exact answers. Ad-hoc queries [Bab+02, Das+07, Gha+08] can be either one-time queries or continuous queries; they are not known in advance; and may require referencing data items that are out-of-scope of the window, and potentially have already been discarded.

3.3.1 Query Semantics

In order to define the semantics and the operators of the DSMS query languages, we first need to introduce two important concepts: *monotonicity* and *non-blocking execution* [Bab+02, ABW06, Tuc+03]. Let $Q(\tau)$ be the result of a continuous query Q at time $\tau \in \mathbb{N}^+$. The correctness of results produced by Q depends on the fact that it will take into account all the data that have arrived so far or is within the window. Since that data change over time within a window, a natural way of switching back to data within a stream is to report the difference between the current result and the result computed one time-tick ago. This leads to the definition of monotonic queries.

Definition 3.4: Monotonic Query

A continuous query or continuous query operator Q is ***monotonic*** *if $Q(\tau) \subseteq Q(\tau')$ for all $\tau \leq \tau'$, where $\tau, \tau' \in \mathbb{N}^+$.*

A simple *selection* over a single stream or a *join* of several streams are monotonic considering that streams are append-only. Thus, when a new data item arrives, it either satisfies the (selection or join) predicate or it does not, and the satisfaction condition does not change over time. Thus, at any point in time, all the previously returned results remain in $Q(\tau)$. On the other hand, continuous queries with negation or set difference are non-monotonic, even on append-only streams.

Definition 3.5: Non-blocking Operator

A continuous query or continuous query operator Q is ***non-blocking*** *if it does not need to wait until it has seen the entire input before producing results.*

The DSMS operators vary around their blocking or non-blocking characteristics, where the blocking operators are not feasible for DSMS queries: blocking operators require to see the whole input streams before generating the output, which is not a viable option for unbounded streams. For instance, traditional SQL queries with aggregation are blocking since they scan the whole relation and then return the answer. However, on-line aggregation, where partial answers are incrementally returned as they are computed over the data seen so far, is considered non-blocking. Note that Definitions 3.4 and 3.5 are related: the class of monotonic queries over data streams can be expressed using only non-blocking operators [LWZ04].

If Q is monotonic, and let $A(Q, \tau)$ be the answer set of Q at time τ, τ_c be the current time ($\tau, \tau_c \in \mathbb{N}^+$), and 0 be the starting time. Then the semantics of an answer set A for a monotonic query Q can be defined as follows [Bab+02]:

$$A(Q, \tau) = \bigcup_{\tau=1}^{\tau_c} (A(Q, \tau) - A(Q, (\tau - 1)) \cup A(Q, 0) \tag{3.1}$$

That is, it suffices to re-evaluate the query over newly arrived data items and append qualifying tuples to the result [GO03]. Consequently, the answer of a monotonic persistent query is a continuous, append-only stream of results. Optionally, the output may be updated periodically by appending a batch of new results. Thus, for a non-monotonic query Q the answers have to be recomputed from scratch with the following semantics.

$$A(Q, \tau) = \bigcup_{\tau=0}^{\tau} A(Q, \tau) \tag{3.2}$$

There are three ways of representing the answer of a non-monotonic persistent query. First, the query could be re-executed from scratch and return a complete answer at every time instant (or periodically). Second, the result may be a materialised view that incurs insertions, deletions, and updates over time. Third, the answer may be a continuous stream that contains new results as well as negative tuples [Ham+03] that correspond to deletions from the result set.

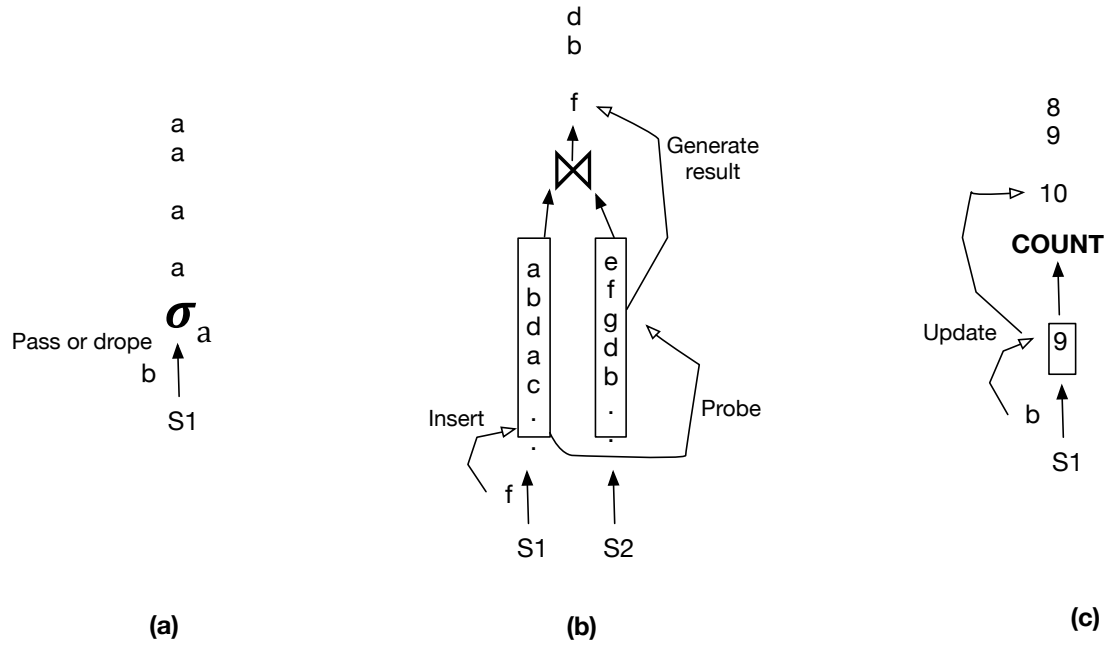

Figure 3.3: Simple Continuous Query Operators: (a) Selection, (b) Join (c) Count (Adapted from [GO03])

3.3.2 Executional Semantics of DSMS

The simplest continuous query operators for DSMS are monotonic; examples include duplicate-preserving projection, selection, and union. These operators process new data items on-the-fly without storing any temporary results, either by discarding unwanted attributes (projection) or dropping data items that do not satisfy the selection condition (technically, the union operator temporarily buffers the inputs to ensure that its output stream is ordered). Figure 3.3(a) shows a simple example of selection (of all the "`a`" tuples) over the character stream `S1`. Figure 3.3(b) illustrates a non-blocking pipelined join of two character streams `S1` and `S2`, where a hash-based implementation maintains hash-tables on both input streams. When a new data item arrives from a stream, it is inserted into its corresponding hash-table and probed against the other stream's hash-table, hence, generating results involving the new data item. Joins of more than two streams and joins of streams with a static relation are straightforward extensions. In the former, for each arrival on one input, the states of all the other inputs are probed in some order. In the latter, new arrivals on the stream trigger the probing of the relation. Figure 3.3(c) shows a `COUNT` aggregate operator. When a new data item arrives, it increments the stored count and appends the new result to the output stream. If the aggregate is based on grouping the results (implemented with a `GROUP BY` clause in the query) it needs to maintain partial counts for each group (e.g., in a hash-table) and emits a new count for a group whenever a new data item with this particular group value arrives. Since aggregates on a whole stream may not be of interest to users, DSMSs support tumbling and/or sliding window aggregates. For efficiency, window aggregates are typically implemented to return new results periodically rather than reacting to each new data item.

3.4 Syntax and Algebraic Properties of DSMS Query Languages

There are three main querying paradigms, described in literature, for DSMSs: relation-based languages [ABW06]; object-based languages [YG02]; and procedural languages [Aba+03]. Here we focus on the relation-based languages, as first they serve as a backbone for the semantically-enabled stream processing; second these languages are frequently used in the general settings for a number of well-known systems. Herein, we present the selected query languages with a focus on Continuous Query Language (CQL), as semantics of most of the existing streaming languages are based on CQL, and are extensions of it.

3.4.1 Continuous Query Language (CQL)

Continuous Query Language (CQL) [ABW06], a declarative query language, is among the first contributions in this field and is considered as an extension of SQL for querying streaming relations. Its main operators, mostly inspired from SQL, include: `SELECT` clause for projection; `FROM` clause to select a specific stream; `PARTITION BY` clause to partition a stream/window on an attribute; window clause (`RANGE`/`ROW`) to specify time/count-based windows; and a `WHERE` clause to define the conditions to be matched by the tuple attributes. Query 3.1 shows a CQL query that uses a stream of sensory information, and computes the average temperature of each sensor located in '`St-Etienne`' with a (tumbling) window of `5 MINUTES`.

```
1 SELECT AVG (S.temp)
2           FROM Sensors [PARTITION BY S.sensor_id
3           RANGE 5 MINUTES] AS S
4           WHERE S.location = 'St-Etienne'
```

Query 3.1: CQL query

In CQL, queries over entire streams may specify `[UNBOUNDED]` or `[NOW]` in the window type, with the latter being used for monotonic queries (e.g. selections) that do not require probing the old items within a stream. CQL is used by the STREAM [Ara+04] DSMS and its abstract semantics are based on two data types: streams and relations, and three classes of operators: *Stream-to-Relation* (sliding windows); *Relation-to-Relation* (corresponding to standard relational algebraic operators); and *Relation-to-Stream*. Conceptually, unbounded streams are converted to relations by utilising sliding windows, The query is computed over the current state of the sliding windows as if it were a traditional SQL query, and the output is converted back to a stream. There are three *Relation-to-Stream* operators: `Istream`; `Dstream`; and `Rstream`. These operators specify the nature of the output. The `Istream` operator returns a stream of all those tuples which exist in a relation at the current time, but did not exist at the current time minus one. The `Istream` operator suggests the incremental evaluation of monotonic queries; and `Dstream` returns a stream of tuples that existed in the given relation in the previous time unit, but not at the current time. Conceptually, `Dstream` is analogous to generating negative tuples for non-monotonic queries. Finally, the `Rstream` operator streams the contents of the entire output relation at the current time and corresponds to generating the complete answer of a non-monotonic query. The `Rstream` operator may also be used in a periodic query evaluation to produce an output stream consisting of a sequence of relations, each corresponding to the answer at a different point in time. The execution of these operators is illustrated as follows.

Consider a case where we have to "continuously" keep track of all the temperature values of all the sensors in "St-Etienne", and report changes in the answer every 5 minutes." Assume that the schema of the input streams consists of three attributes: temp that represents the current observed temperature; sensor_id that represents the id of a sensor; and location that defines the location of the sensor. Then using the stream operator as described above, we can construct the three following queries.

```
SELECT S1.temp
FROM Sensors [RANGE 1 DAY SLIDE 5 MINUTES] AS S1
                    WHERE S1.location = 'St-Etienne'
```

Query 3.2: CQL query for the Relational Output

Query 3.2 describes the situation where the output of situation presented above is a relation and not a stream. The output relation gives the complete query answer and is refreshed every 5 minutes. The output is not incremental, which means that every 5 minutes, the query issuer sees all the temperature values of sensors in the "St-Etienne" location. Now consider another query as follows.

```
SELECT Rstream (S1.temp)
FROM Sensors [RANGE 1 DAY SLIDE 5 MINUTES] AS S1
                    WHERE S1.location = 'St-Etienne'
```

Query 3.3: CQL query for the Rstream Operator

The output of the Query 3.3 is a stream that represents the concatenation of Query 3.2's output relation and it represents the use of Rstream operator. Basically, whenever the output relation is modified (i.e., every 5 minutes, as it is the granularity the window should change), the whole output relation is streamed out (or pushed) to the query issuer. Notice that the output representation is different than Query 3.2, where the output relation is stored and the query issuer needs to pull the modified query answer from the stored relation. Notice also that the output stream, say S_o, is interpreted differently from the input streams. An input data item in input streams (i.e., S1) represents an insertion into the corresponding relations. However, a data item in S_o may represent a repetition for a previous one. For example, temperature values of sensors, that remain constant, for more than 5 minutes are reported several times in S_o.

```
SELECT Istream (S1.temp)
FROM Sensors [RANGE 1 DAY SLIDE 5 MINUTES] AS S1
                    WHERE S1.location = 'St-Etienne'
```

Query 3.4: CQL query for the Istream Operator

The Istream (or insert stream) operator in Query 3.4 produces a tuple in the output stream whenever a tuple is inserted in the output relation (i.e., whenever a sensor reports the temperature value). Notice that because of the slide parameter of length 5 minutes, the inserted data items are accumulated and are produced in the output stream every 5 minutes. Although Istreams's output stream is incremental, it gives only a partial answer because it does not include any information about the temperature values leaving the window: only the difference between the new and older values is added to the output stream.

```
SELECT Dstream (S1.temp)
FROM Sensors [RANGE 1 DAY SLIDE 5 MINUTES] AS S1
                    WHERE S1.location = 'St-Etienne'
```

Query 3.5: CQL query for the Dstream Operator

The `Dstream` (or delete stream) operator, as described in Query 3.5, produces a data item in the output stream whenever a data item is deleted from the relation (i.e., whenever a temperature related data item exits in the window). Notice that because of the slide parameter of length `5`, the deleted data items are accumulated and are produced in the output stream every `5` minutes. `Dstream` output is incremental but it contains partial results (the values that have left the window) from the streams because it does not include the recent information that enter the window.

3.4.2 StreaQuel Language

StreaQuel [Cha+03] is a relational, SQL-derived stream query language and is used in the TelegraphCQ DSMS [Cha+03]: it is noteworthy for its windowing capabilities. The StreaQuel language isolates the streaming semantics from the query language, and the window size used for the query is defined using a for-loop construct. StreaQuel manages unbounded flows by means of its native window operator `WindowIs`; where several `WindowIs` operators may be used in a query, one for each input stream, embedded in a for loop. Each of such operator is part of a rule and defines a time variable indicating when the rule has to be processed. The general assumption behind this mechanism is that it must consider an absolute time model. By adopting an explicit time variable, StreaQuel enables users to define their own policy for moving windows. As a consequence, each window may contain a different number of elements, since its dimension is not bounded a priori. Let `S1` be a stream and let τ_0 be the start time of the query. Then, in order to specify the sliding window that consists of the last `20` time units over the stream, and which runs for `100` time units, the following loop can be used:

```
for ( t = τ0; t < τ0 + 100; t++ )
WindowIs ( S1, t-20, t)
```

StreaQuel inherits SQL operators to define the syntax of stream joins and filtering of attributes: each query is expressed in the SQL syntax and is constructed from the SQL set of relational operators followed by a for-loop construct with a variable `t` that iterates over time. Contrary to CQL, StreaQuel supports both periodic and continuous query processing.

3.4.3 Gigascope Query Language

The Gigascope Query Language (GSQL) [Cra+02, Cra+03] is used in Gigascope, a stream database for network monitoring and analysis. It puts some restrictions over SQL to guarantee that a query cannot produce a non-append-only output. The input and output of each operator is a stream for reasons of composability. Each stream is required to have an ordering attribute, such as timestamp or packet sequence number. GSQL includes a subset of the operators found in SQL, namely selection, aggregation with group-by, and join of two streams, whose predicate must include ordering attributes that form a join window. The stream merge operator, not found in standard SQL, works as an

order-preserving union of ordered streams. This operator is useful in network traffic analysis, where flows from multiple links need to be merged for analysis.

3.4.4 TruSQL

TruSQL [NCT08, Fra+09] is a workflow alike streaming language that extends SQL to query a set of relations as streams. It employs traditional SQL Data Definition Language (DDL) to allow the creation of streams over relational tables. That is by using `CREATE` and `STREAM` operators. Contrary to CQL, TruSQL treats SQL as a first-class concept, and one can pose TruSQL queries directly over streams, tables and combination of streams and tables. A TruSQL query containing no streams is simply a (traditional) SQL query. It inherits many of the pioneering ideas of CQL to reference both streams and relations, such as `Istream`, `Dstream` operators, along-with the introduction of operators such as `VISIBLE` and `ADVANCE`: the `VISIBLE` operator defines the window size and `ADVANCE` operator determines the slide granularity. Thus, they employ different vocabulary for the same kind of operators in CQL with the aim to take it closer to the SQL.

3.5 Existing Data Stream Management Systems

In the previous section, we discussed various query languages for DSMSs that define streaming operators by extending relational languages. In this section, we discuss the underlying architecture of various DSMSs and provide an overview of techniques to efficiently execute the query operators defined in the respective languages.

Traditional DSMS are based on the following main processing blocks: Query Processor, Query Manager, Query Optimiser, Scheduler, Stream Manager, Storage Manager, and QoS monitor. These processing blocks are depicted in Figure 3.4, and in the following we discuss each of them to classify existing systems accordingly.

Stream Manager: As evident from its name, the stream manager provides a set of wrappers that can receive raw data items from its sources, buffer them, order them by timestamps, and convert the rational/semi-structured data items into mapped objects within main memory, such as Java objects.

Queue Manager: The mapped objects are buffered into queues and are handled by the queue manager while utilising a router. The Query manager is highly dictated by the query execution plans, in order to provide the right data to the right query operator. Furthermore, it can also be used to swap data from the queues to a secondary storage, in case the memory resources are exhausted.

Storage Manager: The storage manager is an intermediate block between the queue manager and the secondary storage. It is used when persistent data are integrated with the streaming data; when the streaming data are archived for further processing; or when the streaming data are swapped from the disk to the main memory. Furthermore, it is also utilised to load the parsed query, query plans, etc.

Scheduler: While the queue manager decides according to the query plans which element is processed next, a scheduler determines which operator is executed next. It interacts closely with the query processor which finally processes the query over the arrived data items.

Quality of Service (QoS) Monitor: This component gathers the statistics about performance, order of evaluation query operators, output rate or latency in thought-put.

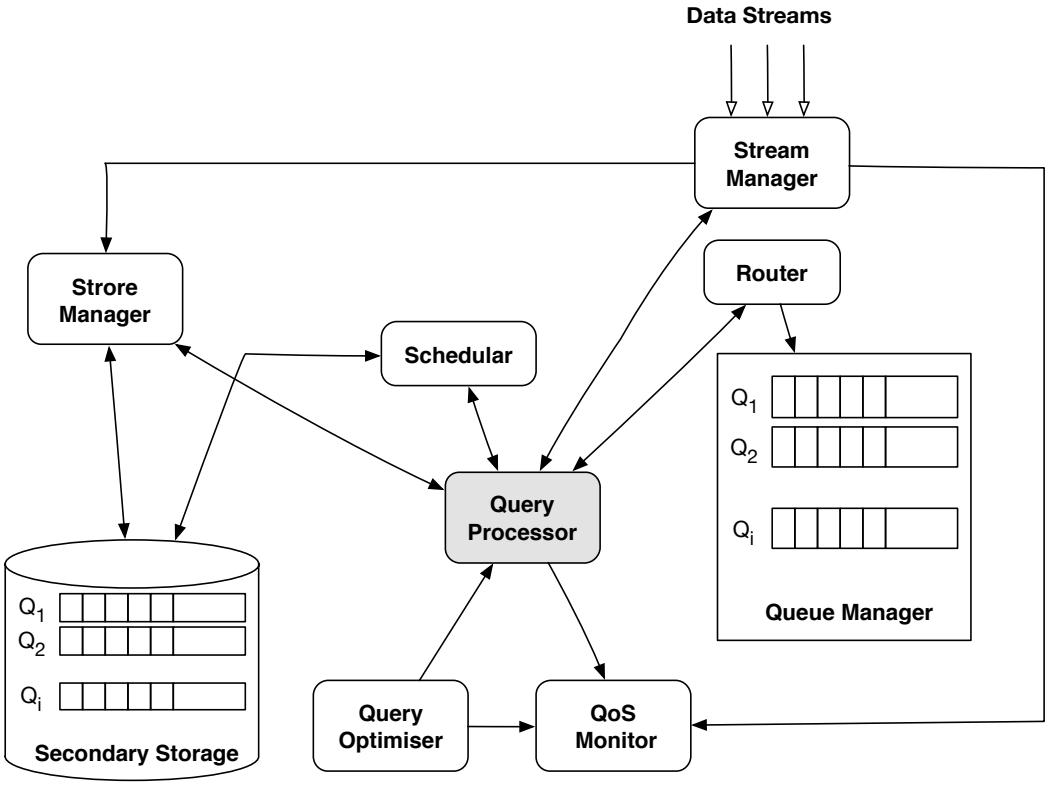

Figure 3.4: A Generic Architecture of DSMS

These statistics can be used during the lifetime of the stream to adaptively improve the query plan and subsequently the system performance.

Query Optimiser: This is the main building block of the DSMS, the differences between most of the DSMSs are mainly based on the implementation of the query optimiser. It directs the scheduler, through query processor, to choose optimal order of execution of query operators, it utilises a load shedder to sample the streams according to the rate of the input, and it employs the plans generated by the QoS monitor. Its main goals include: minimising the computational cost of each query operator, optimising memory usage, flushing older objects from the memory, reducing the size of the intermediate results stored in the main memory. These goals stem from different data handling strategies and are customised by each DSMS.

Query Processor: It provides the harmony between different components of the systems. It itself does not implement any special procedure, instead it relies on the information from the query optimiser, provides current statistics to the QoS monitor, instructs the queue manger to implement the required functionality as directed by the query optimiser.

Based on the general architecture of the DSMS, we describe various optimisations/-customised strategies utilised by existing DSMSs. Note that the following list is not an exhaustive one, instead it contains selected DSMSs related to this thesis.

STREAM [Ara+04]: It is among the first generation of DSMSs, and it uses CQL for its query language. The CQL queries are parsed into tree-based query plans, where each incoming data item is inserted into the query trees. Filtering operators are placed at higher levels within query trees followed by aggregate operators at the leaf level. The

queue router selects the appropriate node within the query-tree and the query processor executes the operators according to the pre-defined query plans; thus the adaptation of the query operators is not tackled. STREAM also provide a customised solution for the memory-limited environments by employing load shedding [Tat+03] on the data items within sliding windows. This technique uses an age-based data arrival model, where the rate at which the data items are processed is solely-based on the age of the data items, which is specified as an age curve.

Aurora/Borealis [Aba+03]: It is developed by a group of researchers from Brandeis University, Brown University, and MIT. Its query language is composed by using operators defined by Aurora Stream Query Algebra (SQuAl): it is a dataflow-like language, where queries are defined using boxes and arcs. Each box represents a query operator and each arc defines the data flow or queue between the operators. Aurora utilised all the main components defined in Figure 3.4, however, most of its optimisations are driven by the QoS component. It defines a number of QoS operators, such as latency-based QoS, loss-tolerant QoS, value-based QoS. Thus, the QoS component directs the query processor to choose the right queue and operator.

Borealis is the commercialised version of Aurora, and inherits most of its optimisation techniques. However, its main focus is on the distributed evaluation of query operators and also on dynamic optimisations to scale with the changing loads and high-availability and resilience against failures.

TelegraphCQ [Cha+03]: It is developed by the University of California, Berkeley with a focus on the adaptive and shared query processing. It employs TruSQL [NCT08], where the commutative query operators are divided into a subset of operators. Operators within each subset are connected to a component called Eddy [AH00], where each Eddy collects the statistic from the QoS monitor to adaptively optimise the execution of query operators. Each subset of operators is processed independently and when all operators are processed within a subset, the intermediate results are routed to the next set of operators or to the output stream.

Other Systems: There are a few other recent systems that follow TelegraphCQ/Truviso's design principle of building a streaming engine out of a relational database engine, however with slight variations of use cases and internal engines. DataCell [LGI09] extends the column-oriented MonetDB relational database for stream processing. Similarly to the STREAM engine, a new datatype called "basket" is introduced in addition to relational tables. Stream tuples are accumulated in baskets and are accessed by continuous queries in a periodic fashion. Baskets allow batch, out-of-order, and shared processing. The general goal of this project is to explore how much the existing relational technology can be exploited for stream processing. As such, it has the potential to naturally integrate DSMS attributes with DBMS ones as the part of its future work. DejaVu [DFT11] provides declarative pattern matching techniques over live and archived streams of events. It extends the MySQL relational database engine and exploits its pluggable storage engine API, where both streaming and historical data sources can be easily attached into a common query engine.

All the above mentioned systems fall under pure relational and homogeneous DSMS, and there is a small body of works that can process heterogeneous streams. Here by heterogeneity we mean that multiple streams with different predefined schemas can be fed into a single system for processing continuous queries. The MDQ (Mapping Data to Queries) [Hen+09] maps incoming data streams of potentially different formats and

schemas to the continuous queries that should process them [Hen+09]. These queries may be written against schemas that are different from the inputs'. MDQ uses a set of schema mapping rules to efficiently decide at run time, which data items should be mapped to which queries. This technique would be quite useful to flexibly process data streams with heterogeneous schemas. Note that, a single query that can integrate multiple heterogeneous streams and process continuously is not addressed in relational DSMSs.

Recently, a new breed of DSMSs has been introduced by integrating Online Transactions Processing (OLTP) and stream processing capabilities [Mee+15, Dug+15]. S-store [Mee+15] provides such capabilities, where an OLTP system (H-store [Kal+08]) addresses the coordination and safety of short atomic computation. It provides the ACID transaction guarantees, and a stream processor to address the needs of real-time applications by providing stream-oriented guarantees. This makes S-store a mutable stream processing system, where the input stream is not append-only, instead a new tuple can be seen as an update to the previous ones. The processing model of such DSMSs is based on dataflow graph, similar to the distributed DSMSs, such as Apache Storm[5], where nodes represent streaming transactions (defined as stored procedures) or nested transactions, and edges represent an execution order [Mee+15]. A set of atomic batches of tuples arrives within streams that are fed to the dataflow graph. With the arrival of new tuples, all the defined streaming transactions defined over the corresponding streams are invoked. The output or processed data items become the atomic batches and are stored for OLTP; traditional window operators are used to constrain the execution of certain stored procedures.

3.6 Optimisation Strategies for the DSMSs

In the presence of commutative query operators, which is quite usually in common cases, DSMS queries can be executed in multiple different ways. It is the responsibility of the query optimiser to enumerate all the possible plans/execution strategies, and to choose an efficient one while considering the cost models and/or set of query transformation rules.

Based on the review of the existing systems, we categorise various optimisation techniques as follow.

Cost-based Techniques: This type of optimisation techniques is based on the selectivity measures of queries and the available indices (i.e., cost models), thus to choose efficient query plans [KNV04]. As these techniques base their roots in static DBMSs, they may not be efficient in streaming settings. However, if stream arrival and output rates are known a priori, it may be possible to find query plans that result in low latency outputs.

Adaptive Query Optimisation: These strategies are based on query rewriting, where the operators (e.g., selection, joins) are executed in a way to minimise the intermediate results. That is, by pushing the least expensive or highly selective operators in the pipeline "on-the-fly". Thus, the re-ordering of the operators is performed on-the-fly in response to the changes in the system conditions [Bab+05]. In this context, Eddy (as described in previous section) performs the scheduling of each data item separately by routing it through the operators that make up the query plans. Consequently, it results in a dynamic re-ordering of the operators to match the stream input and output rates.

Load Shedding and Approximation of Streams: Due to the high stream rates, DSMSs equipped with limited resources may be exhausted and not all the data items can be

[5]Apache Storm: `http://storm.apache.org/` , last accessed: June, 2016.

processed. In such a case, load shedding techniques are applied to drop-off the less significant data items [Tat+03]. The significant measures of data items is measured using their expected expiry time and relevance to produce the expected join results. This, however, can reduce the system's accuracy. Dropping data items based on their significance measures results in effective plans as all the subsequent query operators enjoy reduced loads.

3.7 Summary and Discussion

In this chapter, we provided an overview of the main query languages and the DSMSs that implement them. We reviewed that current DSMSs differ highly in terms of their query languages, semantics and capabilities. Such differences arise due to language choices, targeted use cases and query execution models [Tat10]. The difference in the query syntax and semantics can easily be spotted and handled, whereas differences in the execution models are difficult to handle: they are implicit in the low-level implementation of each DSMS. This requires a coherent formal model that is general and flexible enough to capture and explain the wide range of differences among DSMSs. One effort in this context is carried out by the SECRET framework [Bot+10]. It is a descriptive model that allows practitioners and users to analyse and understand the behaviour of various heterogeneous DSMSs, and provides the conclusive reasons for the variability in the results produced by the DSMS using sliding windows. However, despite these efforts there is neither a common standard for query languages nor an agreement on a common set of operators and their semantics until today. Furthermore, we discuss multiple different types of optimisation techniques employed by DSMS, i.e., cost-based query plans, adaptive query plans, and load shedding.

In the following discussion, we raise some of the expected requirements to consider for the extension or implementation of a DSMS.

Semantics of Computed Results: The explicit description of the semantics of output results is important. CQL provide three main operators including `Istream`, `Dstream` and `Rstream`. These operators are considered as the backbone of all streaming query languages. Thus, a new system/query language should explicitly describe which of these operators are supported.

Choice of the Data model: The choice of the data model can make a huge difference in selecting the optimisation strategies. Most of the systems reviewed in the previous section are based on the relational data model, where schema specific relations are defined within each data item. The simplicity of the data model has encouraged the practitioners to focus on the adaptive optimisations of the DSMSs (i.e., queue manager, QoS monitor, router in Figure 3.4) rather than on how each data item is matched with the query variables. Furthermore, little attention is devoted towards the serialisation of the data items from streams (i.e., stream manager in Figure 3.4). However, such optimisation strategies cannot be applicable in semi-structured/structured data models. For instance, in order to match a data item that is structured with XML, the system would require to consider its tree and cyclic nature.

Incremental Evaluation: The evaluation strategy of stream query operators, i.e., re-evaluation or incremental, is based on the type of the operators as described below:

Selection: The selection operator produces on-the-fly results, and is not bounded by the window operator. Therefore, it is inherently classified as an incremental operator.

Aggregates: Aggregate operators such as *sum, count, average* are non-incremental in nature, and their effective evaluation requires to re-compute the results with the eviction and insertion of new data items within a window.

Joins: The join operators, i.e., window joins or join between relations, also require that the entire window must be probed to get/update the results. Thus, it requires the re-evaluation of all the data items within each window.

Incremental data processing may achieve better performance and may require less memory, as already computed results are reused and merged with the new ones. Recently some techniques, such as exponential histograms [Bra+16], FAT tree structure [Tan+15], are proposed to incrementally compute aggregate operators. These techniques divide the windows into partitions, each storing a partial aggregate. Thus, with the eviction or insertion of new triple partially computed results are refreshed to get the final output. However, there is not much attention towards the incremental evaluation of joins. Few approaches [Ge+15, GGÖ04] are proposed to index each window using a tree structure and join the windows using such indices. However, such indexing techniques are cost-based (as described in Section 3.6) and do not adapt to frequent changes. Nonetheless, these approaches open the door for a new research area.

It is perfectly true, as philosophers say, the life must be understood backwards. But they forget the other proposition, that it must be lived forwards. And if one thinks over that proposition, it becomes more and more evident that life can never really be understood in time; because at no particular moment can I find the necessary resting-place from which to understand it.

— Søren Kierkegaard

4

Semantically-enabled Stream Processing

This chapter paves the way towards the semantically-enabled RDF stream processing. It presents an overview of existing techniques, languages used, and the underlying execution models of RDF stream processing systems (RSPs). While reviewing these techniques, this chapter also highlights various limitations of existing RSP systems. These limitations are addressed by our system called SPECTRA, as described in Chapter 6.

Contents

This chapter is structured as follows: Section 4.1 presents the introductory discussion about the semantically-enabled RDF stream processing (i.e., RSP). Section 4.2 presents the generic data model for RSP systems. Section 4.3 describes the details about the RSP query languages and their execution models. Section 4.4 presents the details of optimisations and execution strategies of RSP systems. Section 4.5 provides an overview of static RDF storage systems. Section 4.6 presents the concluding discussion and lessons learned from the existing RSP solutions.

4.1 Introduction

A number of DSMSs have been developed in the last decade or so (as reviewed in Chapter 3) to tackle the challenges posed by dynamic and high-velocity data streams. However, the issues of handling heterogeneity, integration, and interpretation of data streams at semantic level have been overlooked by DSMSs. The Semantic Web community, through its standards and technologies, is in constant pursue to provide answers to these issues, while employing ontologies, RDF data model, triple stores, etc. The integration of these two contrasting research fields has led to the semantically-enabled stream processing, usually called *RDF Stream Processing* (RSP) [Bar+10b, LP+11, CMC16, KCF12].

RSP systems employ the RDF model, where streams consist of an infinite sequence of RDF triples each associated with a timestamp. Currently, the W3C community group, called RSP community group[1], is working towards the standardisation of the following concepts:

- RDF stream model, it was first introduce in [DV+09], and later picked-up in [Bar+10a, LP+11].
- Extension of SPARQL to enable continuous query processing, such as C-SPARQL [Bar+10b], CQELS [LP+11].

Despite the recent efforts, there does not exist a standardised query language of RSP streams. Hence in the proceeding section, we first introduce the RDF stream model, which is employed by most of the existing RSP systems, and query languages for RSP systems.

4.2 RSP Data Model

RSP systems have integrated temporal attributes with each RDF graph triple for their data models, where an RDF triple, as described in Definition 2.2, is a tuple $t \in (\mathcal{I} \cup \mathcal{B}) \times \mathcal{I} \times (\mathcal{I} \cup \mathcal{B} \cup \mathcal{L})$. Then an RDF stream [Bar+10a, LP+11] can be defined as follows.

Definition 4.1: RDF Stream

*An **RDF stream** S_r is a sequence of pairs (t, τ), where t is an RDF triple $\langle s, p, o \rangle$, and τ is a timestamp in the infinite set of non-decreasing timestamps $\mathbb{T}$, such that*

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

An RDF stream is an append-only stream, where SPARQL graph patterns are used to match a set of streams. The execution of the graph patterns over the RDF streams is constrained by the standard window operations as described in Definition 3.2. The RDF stream model can be seen as a direct extension of the relational data stream model, where an RDF triple is used instead of a relation tuple. Hence, the languages and the operators proposed by the RSP systems also inherit from DSMS's operators.

[1] W3C RSP Working Group: `http://www.w3.org/community/rsp/`, last accessed: June, 2016.

4.3 RSP Systems and Their Query Languages

SPARQL (see Chapter 2) is a standard RDF language, and similar to SQL it is designed for on-shot queries. A number of RSP languages are proposed to extend SPARQL with operators that take into account the streaming nature of RDF streams. The two most common languages in this context are C-SPARQL [Bar+10a, Bar+10b] and CQELS [LP+11]. These languages, similar to CQL, integrate the *window* and `FROM STREAM` operators to define query graphs over a set of RDF streams, and to process them in a continuous manner.

4.3.1 C-SPARQL

C-SPARQL [Bar+10a, Bar+10b] is among the first contributions in the area of RSP, and is often cited as a reference work in this field. The distinguished features of C-SPARQL are described as follows:

1. support of RDF stream model,
2. support of defining a set of streams over a set of triple patterns,
3. support of aggregate operators, and
4. support of static data and streaming data integration.

C-SPARQL borrows the concept of windows from DSMSs (in particular CQL) to capture the portions of each stream that are relevant for processing: it can be either count-based (selecting a fixed number of triples) or time-based (selecting a variable number of triples which occur during a given time interval). Furthermore, additional policies of windows such as *slide* are also supported.

Thus, the set of triples within a window is matched against the defined query graphs for each execution of the C-SPARQL query. It allows, similarly to DSMSs, to register continuous queries: queries are issued once against the set of RDF streams and continuously evaluated, and the matched mappings (see Chapter 2) are composed by listing the matches of each evaluation.

```
1 SELECT ?temp ?id
2 FROM STREAM <http://streamsensor.com/source1> [RANGE 3s STEP 1s]
3 FROM STREAM <http://streamsensor.com/source2> [RANGE 3s STEP 1s]
4                 WHERE {
5                         ?source :sensor_id ?id.
6                         ?source :location  ?loc.
7                         ?source :temperature ?temp.
8                         Filter (?loc = 'St-Etienne')
9                         }
```

Query 4.1: C-SPARQL query

Query 4.1 presents a simple C-SPARQL query to obtain the temperature and sensor id values from two different RDF streams. It uses a `FROM STREAM` operator to register RDF streams: it could be a set of heterogeneous RDF streams. Similarly to CQL the operators `RANGE` and `STEP` define the sliding window, i.e., defining the size (`3s`) and the granularity it slides with (`1s`). Note that, a `FROM` clause can be defined in C-SPARQL to register a static background knowledge-base.

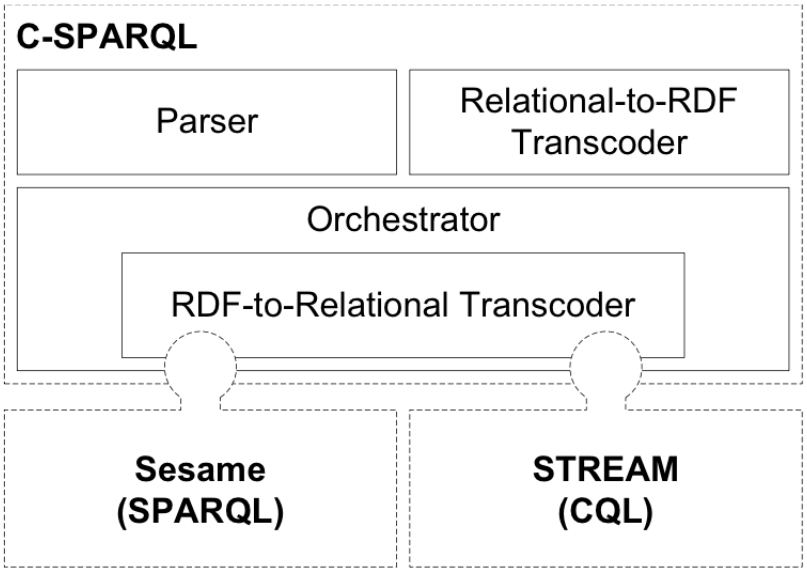

Figure 4.1: CQELS Architecture (adapted from [Bar+10a])

Evaluation of C-SPARQL Queries: C-SPARQL employs a "black-box" approach to delegate the execution of queries to the underlying DSMS (i.e., Esper [BV10] or STREAM [Ara+04]) and an RDF engine/store (Jena or Sesame). Each C-SPARQL query is transformed into an O-GRAPH [Bar+10a], where the static and streaming parts of the query are mapped into relational bindings. The C-SPARQL engine orchestrates the execution of queries, while employing a DSMS and an RDF store against these relations.

Figure 4.1 shows the architecture of CQELS using STREAM and Sesame [BKH02]. Each query is parsed and assigned to the orchestrator, which translates it into static and streaming parts. The static query is used to extract static knowledge from the triple store, while the dynamic part of the query is registered in the DSMS. When translating C-SPARQL queries into static and streaming parts, the orchestrator relies on the information captured by the so-called Denotational Graph or D-Graph in order to distinguish static from streaming knowledge [Bar+10a]. Although various optimisation techniques, such as pushing the selection, filter and aggregate operators at the top of the query execution stack, are proposed for C-SPARQL, it is not a performance intensive engine. Semantic-wise it is a pull-based system, where the matches are produced periodically: it employs the `Rstream` operator from CQL. More discussion will follow later on this point in Section 4.4 and 4.6.

4.3.2 CQELS

CQELS [LP+11] is another RSP query language inspired from CQL. It also provides continuous RSP and utilises static background data to enrich RDF streams. As far as the syntax of the language is concerned, it employs the similar query constructs as those of C-SPARQL. However, its `STREAM` clause wraps the defined graph patterns for each stream: it is analogous to the `GRAPH` clause of SPARQL.

```
SELECT ?temp ?id ?temp2 ?id2

    WHERE {

     STREAM <http://streamsensor.com/source1> [RANGE 3s STEP 1s]
                 {
                       ?source :sensor_id ?id.
                       ?source :location  ?loc.
                       ?source :temperature ?temp.
                       Filter (?loc ='St-Etienne')
                  }
    STREAM <http://streamsensor.com/source2> [RANGE 3s STEP 1s]
                  {
                       ?source2 :sensor_id ?id2.
                       ?source2 :location  ?loc2.
                       ?source2 :temperature ?temp2.
                       Filter (?loc2 ='St-Etienne')
                  } }
```

Query 4.2: CQELS query

Query 4.2 describes the same use case as discussed above. The window content is accessed within each `STREAM` clause and users can define different types of windows for each stream, a selling point of CQELS language.

Evaluation of CQELS Queries: Contrary to C-SPARQL, CQELS uses a "white-box" approach porting DSMS concepts (e.g. physical operators, data structures and query executor) into an SPARQL engine. For the underlying DSMS, it also employs Esper [BV10]. The evaluation semantics of CQELS are push-based, i.e., the evaluation of the queries is triggered with the arrival of each new triple within streams, and only newly produced matches are added to the output stream. Thus, it employs the `Istream` operator [ABW06] of CQL. Due to its customised architecture, CQELS is much more performance competitive as compared to other RSP engines. This is due to its reliance on the existing optimisations for DSMSs: it employs adaptive reordering of query operators, while utilising a query monitor[2] (through Eddy operators [AH00]) to push the less expensive operators in front of the query plans.

CQELS query plans are constructed through the data flows [Aba+03], where a data flow constitutes a directed tree of operators. The root of the tree is either a relational or a streaming operator, while leaves and intermediate nodes are window and relational operators respectively. Figure 4.2 shows a data flow for Query 4.2: the graph pattern P_1 is the defined set of triple patterns for the stream S_1, and the graph pattern P_2 is the defined set of triple patterns for the stream S_2. The matched results of each operator are joined together and sent to the output streams S_{out}. During the whole lifespan of the streams, the CQELS engine constantly attempts to determine the optimised order of data flow operators for an optimised query execution.

4.3.3 StreamQR

StreamQR [CCG10] offers a different view for RSP systems by providing a streaming system equipped with inference capabilities. The goal of the system is to employ query

[2] As discussed in Chapter 3 (Section 3.5), the QoS monitor partition the commutative query operators, where each subset of the operators are executed independently and the intermediate results are fed to the remaining partitions through Eddy operators.

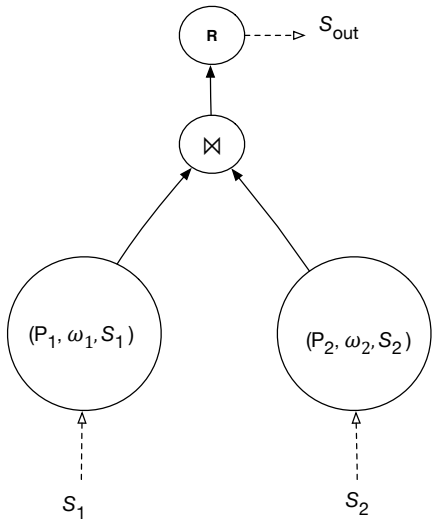

Figure 4.2: Data flow in CQELS for the Query 4.2

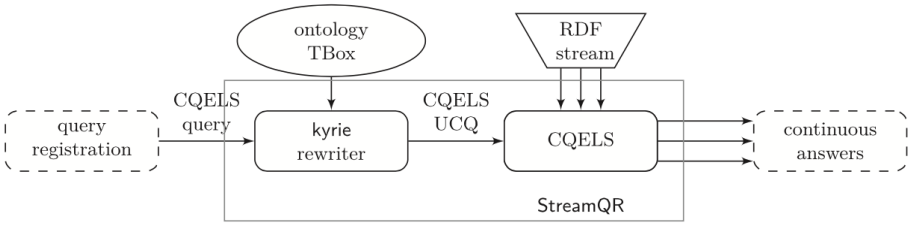

Figure 4.3: Architecture of StreamQR (adapted from [CMC16])

rewriting to transform continuous queries (with CQELS) into an expanded query that captures the ontology TBoxes [Cal+07], i.e., intensional knowledge. The expanded query (or a set of sub-queries) are evaluated over ABoxes, i.e., extensional knowledge. This approach can be equated with the *ontology-based data access* [Bie+14], where queries are rewritten through an ontology to query data stored in relational databases. However, in this context, rewritten queries are computed over the RDF streams. This work does not provide a new system or optimisation techniques for RSP, instead it coupled CQELS with a query rewriter called kyrie [MC13] to enable such functionality.

Figure 4.3 shows the architecture of StreamQL. The input CQELS query is fed into the Kyrie rewriter, which uses $\mathcal{ELHIO}$ [PUHM09] as a language for the ontology to rewrite the graph patterns within the CQELS query to produce a union of conjunctive queries (UCQ). The UCQ are synthetically transformed back to a CQELS query using context information from the original query, i.e., window and stream definitions. Finally, the transformed query is evaluated by the CQELS engine over the RDF streams, and the matches are appended into the output stream.

4.3.4 Sparkwave

Sparkwave [KCF12] is a recent solution that utilises the RETE [For90] algorithm to continuously process the RDF streams against the defined C-SPARQL queries. RETE [For90] is a rule-based algorithm, and is based on forward chaining and inferencing of facts and

rules. The RETE algorithm builds a directed acyclic graph as a high-level representation of the given rule sets; these are generated at run-time and include objects such as nodes of the network. Each rule in RETE is processed in three stages: match, select and execute. In the first stage, the conditions of the rule set are matched against the facts to determine which rules are to be executed. The rules whose conditions are matched are stored in an agenda list to be fired. From the agenda list, rules are selected and executed depending upon a priority, recency of usage, specificity of the rules, or on other criteria. The rules are executed by exciting the actions defined in the rules. The nodes in the RETE network, i.e., the network of rules, are of three types, alpha (α), beta (β) and terminal node. Each α node represents the match node, in order to match the antecedents defined in the rules, while each β node, also called merge node, merge two or more α nodes. The terminal node contains the consequent of the beta nodes. For instance, take a composite rule, $A > B$, then two alpha nodes are created, one for A and another for B, while a beta node is created that will check the conditions to be matched between A and B (see Figure 4.4). The match of beta node is fed to the terminal node for the output.

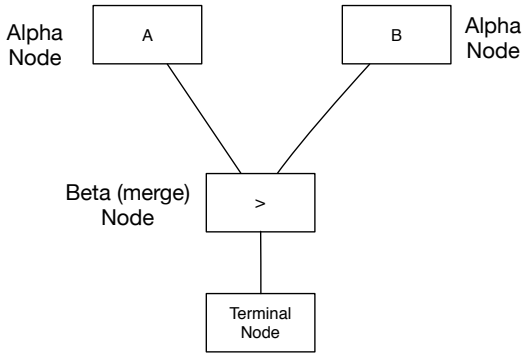

Figure 4.4: RETE nodes for the rule $A > B$

Sparwave is designed to provide a high-performance stream processing, where streams are defined explicitly through a schema-entailed knowledge. Sparkwave includes limited support for background knowledge (schema and static data instances) and supports only a limited set of schema constructs; it is therefore complementary to other solutions which offer such functionalities but in the context of less stringent performance requirements. It does not provide a new language for RDF streams, but it employs the C-SPARQL query language on top of its RETE network of α and β nodes: the registered query is parsed into a set of streaming rules that sits in the production memory, and a set of facts (background knowledge) in the data memory. These rules are defined as α and β nodes in the RETE network. The α nodes use the filter, projection and join attributes, defined in the query graph, to check if the specific conditions are fulfilled by the streaming triples. The β nodes store the intermediate joined results in form of tokens [KCF12]. A token k is a pair (k_{parent}, st), where k refers to a parent token and st as a streaming triple that is stored in an α node.

4.3.5 Other Systems

There are few more solutions provided in the context of RSP. They either employ ontology-based data access to directly a query relational data stream with the aid of an ontology, or only provides the theoretical details of RSP languages without any explicit execution model.

SPARQL$_{stream}$ offers a different view of RSP systems. It provides a streaming solution for ontology-based data access, where the registered query, an extension of SPARQL, is mapped onto a set of SNEEql [Gal+09] subqueries to be processed against the data streams. It does not provide a new streaming framework, but uses the ODEMAPSTER

processor [GJS92a] to process SNEEql queries (as discussed in Chapter 3).

Streaming-SPARQL [BGJ08] presents an extension of SPARQL to process RDF streams, and offers a theoretical view. The main focus of the work is to provide the semantics of the new streaming operators, and how to translate general SPARQL queries into streaming ones with extended algebra.

RSP-QL [Del+14] provides an abstract query model to expose the heterogeneity among various RSP query languages, in particular C-SPARQL and CQELS. Such a formal model characterised RSP systems for defining interoperability among RSP systems, and for defining the correctness of query results. Although, it does not provide an explicit query language or execution model, it outlines the semantic heterogeneity among RSP systems.

4.4 Under the Hood of RSP Systems

The evolution of RSP systems from the DSMSs somehow has undermined the graph nature of RDF streams. That is, contrary to relational-based DSMS where a relational tuple contains a set of attributes – each describing an object value mapping – an RDF stream consists of triple-based elements. Thus, each RDF triple is only able to describe a single subject to object relation through a predicate. This results in a potentially large number of RDF triples, each being processed as a new element within a stream. For instance, if there is a stream containing values emitted by a sensor, each data item for a relational stream will consists of a set of objects and values, such as temperature, sensor-id, location, etc. However, in the case of RDF streams, each stream element consists of a single triple, mapping a single attribute such as temperature value or sensor-id. Therefore, instead processing each data item containing a bulk of attributes, each triple containing a single attribute is processed independently. Due to the large number of triples within a window, this requires careful consideration when implementing the joins between the query triple patterns. However, existing RSP systems do little to nothing in this context. They reuse and adapt indexing techniques from the static RDF solutions that are prone to frequent updates, and operator re-ordering techniques from DSMSs that choke under large windows. We briefly discuss the details of the underlying architectures for the two well-known RSP engines CQELS and C-SPARQL; such discussion sheds some light on the improvements that can be incorporated in our system design (as discussed in Chapter 7).

C-SPARQL, as discussed before, completely ignores the optimisation techniques for indexing and query RDF streams. It is based on the static RDF triple store Jena [Car+04] (older version utilises Sesamse [BKH02]), which employs property tables to store triples and a B+ tree indexing to guide the query process. For the same reason, many studies show that it is not scalable when the size of the window is increased to thousands of events. Thus, for each new RDF triple within a stream, it is added to the property table and an appropriate index is inserted in the B+ tree. The same procedure is applied when a triple is removed from the window. When a query is executed over a set of triples, all the matches are reproduced and sent to the output stream. This not only results in frequent insertions and deletions from B+ tree, which is computation intensive considering the number of such operations, but also the recomputation of query matches from scratch. These are the primary seeds of improvements to scale up such RSP systems.

CQELS improves on few of the C-SPARQL points. It provides various adaptive optimisations to join the set of triple patterns and uses the `Istream` operator to output only the changes in the matches. However, such optimisations are again inspired from the relational techniques that are not quite suited for graph-structured streams. For instance,

Table 4.1: Classification of Existing RSP Systems

RSP Systems	Input Model	Execution Model	Background Knowledge	Time Model
C-SPARQL	Triple Streams	Pull-based	✓	Timepoints
CQELS	Triple Streams	Push-based	✓	Timepoints
StreamQR	Triple Streams	Push/Pull-based	✓	Timepoints
SparkWave	Triple Streams	Push-based	✓	Timepoints

Table 4.2: Classification of Existing RSP Systems

RSP Systems	Reasoning	Temporal Operators	Historical Data and Statefulness
C-SPARQL	RDFS subset	✗	✗
CQELS	✗	✗	✗
StreamQR	$\mathcal{ELHIO}$	✗	✗
SparkWave	RDFS subset	✗	✗

Eddy operators [AH00] improve performance by re-ordering the join and filter operators between multiple RDF streams, however, they suffer from performance degradation if applied frequently over a larger number of triples. Such is the case of RDF streams, where usually the window contains a large number of triples instead of a relatively small number of data items. The indexing strategy of CQELS is again based on the traditional B+ trees, which are not friendly for frequent insert and delete operations. As a result of these shortcomings, there are large performance differences between CQELS and the DSMSs.

Table 4.1 and Table 4.2 summarise various attributes and capabilities of existing RSP systems as described earlier. That is, all the RSP system are based on timepoints semantics, and their execution models are either push or pull-based. Furthermore, they all employ simple triple streams, where each incoming item consists of a triple associated with the timestamp. From the reasoning aspect, C-SPARQL and SparkWave support the subset of RDFS rules to infer information from streams, while StreamQR use a dedicated OWL reasoner for a broad set of complex rules.

In this section, we outline the inner working and attributes of the existing RSP systems. In order to provide the overview of the graph pattern matching techniques utilised by the RSP engines, we summarise various static RDF graph stores and their corresponding attributes in the proceeding section.

4.5 RDF Graph Storage and Processing Techniques

In the previous sections, our discussion was mainly focussed on the RSP systems. In this section, we briefly discuss selective static RDF storage and querying systems.

Most of the existing solutions for querying static RDF graphs, that differ in the underlying storage structure and the type of indexing, match a query with the RDF dataset in two steps. The first step retrieves a candidate set of graphs that contains the indexed features of the query. The second step uses subgraph isomorphism (subsequently homomorphism) to validate each candidate graph against the defined query. In general, the RDF graph storage systems – both in-memory and disk-based – can be classified

into native and non-native RDF storage systems. The native solutions consider the RDF data model as a first class citizen and provide customised methods (with customised storage and indexing techniques), while the non-native solutions borrows data storage models from the DBMS and customise them with further indexing techniques. Some of the major works in this context are summarised as follows.

4.5.1 Native RDF Graph Storage Systems

Most of the native RDF graph storage techniques eschew the mapping to an RDBMS and focus instead on indexing techniques specific to the RDF data model. Thus, these approaches are based on sophisticated indexing techniques that are customised according to the RDF data model.

RDF3x [NW10b]: It is the most prominent solution in this context, and it employs six different types of indices over multiple redundant $\langle subject, predicate, object \rangle$ permutations. It creates its indices over a single "giant triples table", and stores them in (compressed) clustered B+ trees. Triples, within each index, are lexicographically sorted allowing SPARQL patterns to be converted into range scans. The triple store is compressed by replacing long string literals in the triples IDs using a mapping dictionary. RDF3x reports a very efficient performance that outperforms other RDF stores by an order of magnitude [NW10b]. These results make it a leading reference in this area. However, despite its compression achievements, the spatial requirements in RDF3x remain very high. This involves an indirect overhead to the querying performance because large amounts of data need to be transferred from disk to memory, and this can be a very expensive process with respect to the query resolution itself.

BitMat [Atr+10]: It follows the idea of managing compressed indices but it goes another step further and proposes querying algorithms that directly perform on the compressed representation. BitMat introduces an innovative compressed bit-matrix to represent the RDF structure. It is conceptually designed as a bit-cube ($s \times p \times o$), but its final implementation slices to get two-dimensional matrices: *so* and *os* for each predicate p, *po* for each subject s, and *ps* for each object o. These matrices are run-length compressed by taking advantage of their sparseness [Atr+10]. Two additional bit-arrays are used to mark non-empty rows and columns in the bitmats *so* and *os*. The results reported for BitMat show that it only overcomes the state of the art for low selectivity queries.

Hexastore [WKB08]: It is based on the idea of main-memory indexing for RDF data in a multiple-index framework. The RDF data are indexed in six possible ways, one for each possible ordering of the three RDF elements by individual columns. The representation is based on all the possible orders of significance of RDF resources and predicates, and can be seen as a combination of vertical partitioning [Aba+07] and multiple indexing approaches [HD05]. Two vectors are associated with each RDF element, one for each of the others two RDF elements (e.g., (*subject, predicate*) and (*subject, object*)). Moreover, lists of the third RDF element are appended to the elements in these vectors. Hence, a *sixtuple* indexing schema is created. However, even though six tables are created, only five copies of the data are really computed, since the object columns are duplicated. Hexastore provides efficient single triple pattern lookups, and also allows fast merge-joins for any pair of two triple patterns. However, space requirement of Hexastore is five times the space required for storing statements in a triple table.

RDFox [Nen+15]: It is an in-memory triple store with the emphasis on the multi-

threaded lock-free access of materialised triples. RDFox stores the triples as a *TripleList* in one big-table called as *TripleTable*; along-with the indices that support the iteration over subsets of the triples. A TripleList stores the RDF triples as a two-dimensional array with six columns: the first three columns hold the IDs of the subject, predicate, and object of a triple, while the latter three columns are used for indexing. In particular, the triples in the TripleList are organised in three linked lists, each of which is grouped by subject, predicate, and object, respectively. Thus, the last three columns in the TripleList provide the next pointers in the respective lists. These linked lists are used to efficiently iterate over triples matching a combination of subject, predicate, and object. It also employs B+ tree indexing to enable cyclic and complex queries on top of its underlying storage structure. Due to its lock-free and parallel architecture, RDFox is performance intensive. Although, there does not exist any comparative analysis, it can materialise a LUMB-5000 [GPH05] dataset in 422 seconds using a single core, and this reduces to only 42 seconds when using 16 cores.

4.5.2 Non-Native RDF Graph Storage Systems

The non-native RDF solutions are inspired and extended from well-established relational DBMSs. These solutions store RDF triples into a set of relational tables, of different types, while building indexing on top of them to support SPARQL queries. The two most important techniques in this context are *property tables* and *vertically-partitioned tables*.

Property Tables: This approach creates relational-oriented property tables out of RDF data, where each table gathers the information about the multiple predicates/properties over a list of similar subjects/objects. Each property table contains multiple different columns, since different predicates (one per column) are used for describing the subjects it stores (in rows). Although, this model significantly reduces the number of self-joins (i.e., the relationship between rows stored in the same table), the cost of query operations remains high due to redundant query operations. Furthermore, this technique increases the storage and querying cost (1) by explicitly storing NULL values in each subject if the represented subject is not described for a given property in the table; (2) by its inability to handle multi-valued attributes that are abundant in the RDF dataset. Systems like Jena [Car+04] and Sesame [BKH02] utilise the property tables as their underlying data structure and B+ trees for the indexing of triples.

Vertically-partitioned Tables: The vertical partitioning (VP) approach [Aba+07, Sub+16] can be seen as a specialised case of property tables, where each table assembles information about each distinct predicate in the RDF dataset. Thus, the VP approach creates as many distinct tables as the number of distinct predicates within an RDF dataset, each containing two columns (subject, object) to stores all the subjects and objects relating to a particular property. The VP approach covers all the issues surfaced by property tables including: NULL values and multi-valued attributes. Each VP table is usually sorted on the subject column to enable fast merge join to reconstruct information about multiple predicates for the subsets of subjects. Systems such as SW-store [Aba+09a] utilise VP tables along with the B+ tree indexing to support multi-join SPARQL queries.

4.6 Summary and Discussion

How to build an optimise RSP system? In this chapter, we reviewed the existing RSP systems and selective static triple stores to answer such question. We provided the

Table 4.3: Optimisation and Underlying Engines for RSP systems

RSP Systems	Underlying Engine	Optimisations
CQELS [LP+11]	Esper	Adaptive reordering of query operators
C-SPARQL [Bar+10b]	Esper and Jena	Pushing filter and selection expression
SparkWave [KCF12]	RETE	Streaming α and β nodes
StreamQR [CMC16]	CQELS and kyrie	Rewriting of CQELS queries

basic model for RDF streams and discussed the existing extensions of SPARQL to enable continuous query processing. Furthermore, we also reviewed the existing static triple stores on the basis that RSP systems directly inherit their storage and indexing expertise from these systems.

Based on our analysis, our guide to the practitioners is as follows:

- *RDF Graph Model for Stream Elements*: Most of the existing RSP solutions are based on a RDF stream model, where each element within a stream is a triple associated with a timestamp. However, real-world use cases differ from it. For instance, a sensor usually emits a set of environmental attributes at the same time. Considering this, we recommend the use of an RDF graph-based model for RDF streams, i.e., a set of triples for each element of the stream (more discussion on this point can be found in Chapter 6 and Chapter 8). Furthermore, the insertion and eviction of a single triple, as compared to a set of triples, may increase the load over the query processor: an empirical analysis to support this hypothesis is provided in Chapter 9. Moreover, the use of an RDF graph model is also recommended by the W3C RSP community group[3].

- *Customised Optimisation for RDF Streams*: Under the hood, most of the RSP engines are based on the techniques directly borrowed for the DSMSs and static RDF triple stores; table 4.3 summarises the underlying optimisation techniques and engines for RSP systems. This raises certain questions about their performance and scalability. Therefore, we recommend to take the RDF graph model as first class citizen while providing customised optimisations for RDF streams. That is, (i) reordering query operators according to the structure of the query graphs (star, chain, complex-shapes) and the incoming events, (ii) incremental indexing for the graph structured data which is not prone to frequent insertion and deletions.

- *Considering Heterogeneity among RSP Systems*: All of the above mentioned systems are heterogeneous in nature in terms of their proposed languages, executional models and way they report the results. Some of the differences in the RSP engines are reflected in how the query dataset is constructed and how windows are declared. For instance, CQELS associates a named (time-varying) graph to each window in the query, and the window content is accessed with the `STREAM` clause, analogous to the `GRAPH` in SPARQL. However, it is not possible to declare the sliding window in such a way that its content is included in the default graph of the dataset. On the contrary, C-SPARQL does not allow to name the time-varying graphs computed by the sliding windows, but all the graphs computed by the sliding windows are merged and set as the default graph. Furthermore, CQELS employs the `Istream` streaming operator, while C-SPARQL is based on `Rstream` operators. Therefore,

[3]W3C RSP Community Group: `https://www.w3.org/community/rsp/`, last accessed: June, 2016.

while comparing and implementing an RSP system, one should carefully consider the differences between these systems.

As a contribution of this thesis, we propose new optimisation techniques for processing RDF graph streams, while learning constructive lessons from the existing solutions. Our solutions consider not only the complex graph nature of RDF, but also the constraints imposed by streaming settings. This discussion and proposed solutions are provided in Chapters 6 and 7.

Nothing is more usual and more natural for those, who pretend to discover anything new to the world in philosophy and the sciences, than to insinuate the praises of their own systems, by decrying all those, which have been advanced before them.

— David Hume, *A Treatise of Human Nature*

5 Detection of Complex Event Patterns

This chapter provides the introductory details for the second phase of our work: semantically-enabled pattern matching or Semantic Complex Event Processing (SCEP). We first discuss the relational approaches for Complex Event Processing (CEP), and outline the properties, data models and languages used in the existing works. Since CEP is a field that is very broad and without clear-cut boundaries, this chapter focuses strongly on the topic of this thesis. That is, on querying complex events and extending it for SCEP. It concentrates on languages and execution models for detecting complex events that are known and specified a priori.

Contents

This chapter is structured as follows: Section 5.1 provides introductory remarks about CEP. Section 5.2 presents data models for CEP. Section 5.2.2 presents various query languages and their operators for CEP. Section 5.3 provides a detailed analysis of models and techniques used by existing CEP systems. Section 5.4 presents a detailed analysis of existing SCEP techniques. Section 5.5 concludes the chapter with a discussion.

5.1 Introduction

A Data Stream Management System (DSMS) works on an unbounded sequence of time-stamped data items, where data items within streams and bounded by the windows are matched against query-defined aggregates and filtering operators. The notions of temporal statefulness between data items, which represents the temporal relations, was not made explicit by the DSMSs and is ignored by system architectures. In the context of Complex Event Processing (CEP), an event typically corresponds to a single data item with temporal attributes and a defined temporal pattern matches not only the data values within data items, but also the temporal relations between a set of events.

Consider the sensor networks example, where a user would like to capture the temporal pattern of temperature values (reported by a sensor). That is, a pattern describing that the temperature has decreased to a local minimum and then raised to a maximum value notifies a specific event happening in the external world. Supporting such attributes requires careful consideration of the temporal properties of data items, and the statefulness within data items. Hence, for each execution of the system, the newly arrived data items are not only matched with the query-defined filters, but also with the query-defined temporal properties (sequences). These characteristics are covered under the umbrella of CEP [CM12]. CEP model views flowing information items as notifications of events happening in the external world, which have to be filtered and combined to understand what is happening in terms of higher-level events [CM12].

The origins of CEP approaches may be traced back to the publish-subscribe domain [Eug+03]. While the traditional publish-subscribe systems consider each event separately from the others, and filter them (based on their topic or content) to decide if they are relevant for subscribers, CEP systems extend this functionality by increasing the expressive power of the subscription language to consider complex event patterns that involve the occurrence of multiple related events.

There are several definitions of CEP applications [Bre+09, Luc01, WDR06a], but they commonly involve three requirements:

1. complex predicates (filtering, correlation),
2. temporal/order/sequential patterns,
3. transforming the event(s) into more complex structures.

Herein, we first describe the query languages and systems for CEP and then move towards the semantically-enabled CEP. For brevity, a discussion of the following topics has been omitted: these are not directly related to the topic of this thesis, and are considered as customised use cases.

1. CEP over out-of-order streams or streams with imprecise timestamps, such as works presented in [FAR11, ZDI10].
2. Distributed CEP, such as works presented in [Bre+09, SMMP09].

5.2 Data Model and Operators for Complex Event Processing

CEP has evolved from many different research areas, thus a standard terminology has not yet established and found broad adoption [Eck+11]. For example, what is called a

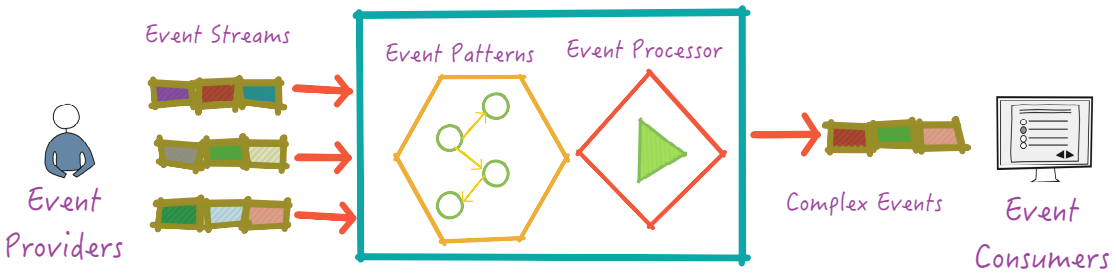

Figure 5.1: High-level Overview of CEP System

(*complex*) event query might also be called a *complex event type*, an *event profile*, or an *event pattern*, depending on the context. Therefore, before diving into matters it is worth to settle some terminology and the basic functioning of event queries, and to make it more precise.

Event queries are evaluated over time in a system called the *event processor*, event query evaluation engine, or CEP engine. Its *inputs* are data items called *simple events*. An *event* indicates that something of interest has happened or is contemplated as happening. Events can be represented in many data formats, such as relational tuples, XML documents or objects of an object-oriented language. Similarly to other types of data, it is common to classify events according to some *type*: for XML it can be the schema, or an explicit ID for the relational tuples.

Since simple events are received over time, the input takes the form of one or more streams. Every simple event is associated with at least one *time point or interval* called its *occurrence time*. Unless otherwise specified, we assume here for simplicity of presentation that events have only one occurrence time. The output of the event processor are the query answers or complex events. The output typically takes the form of a new data object (e.g., a message to be communicated to another system). However, the term output should be understood broadly to include for example also the cases where the event processor does not explicitly construct new data but directly initiates some action, e.g., an update to a database or displaying something in a graphical user interface.

5.2.1 Data Model

Herein, we extend the Definition 3.1 to formally describe an event and event stream.

Definition 5.1: Relational Event and Relational Event Stream

A `Relational event`*, or simply an event, e is a pair (v, τ), where $v \in \Re$ is a relational tuple and $\tau \in \mathbb{N}^+$ is an associated timestamp. A* ***relational event stream*** *$\mathcal{S}_e$ is a countable infinite set of events.*

The relational tuple (v) within an event (e) consists of a set of attributes, i.e., an entry of a tuple and the set of values of the event attributes: it is called *event data*. Since an event happens at a particular time, the implicit timestamp of an event is called *event occurrence time*. Note that the above definition for an event can easily be extended, where event occurrence time is captured by a time-interval, i.e., using two timestamps to indicate its bounds.

Figure 5.1 illustrate the high-level architecture of a CEP system, where events are sent by the event producers (e.g., sensors) to event processor (CEP system) as an event

stream. The event processor utilises an event query language (EQL) which matches events to its described types in the query languages. For instance, `highTemp(area (a))` is an event defined in an EQL indicating high temperature in an area `a`. A simple query is a specification of a certain kind of single event, while a complex query specifies certain combination of events with multiple event queries and the conditions describing the correlation between them. For example, `highTemp(area (a))` $\rightarrow$ `smoke (area (a))` is a complex event that could be interpreted as the presence of fire. That is, if high temperature event is followed by ($\rightarrow$) smoke event in an area.

5.2.2 Event Query Languages and their Operators

Complex event queries are associated with actions that are to be performed whenever a complex event is detected. Since actions are sensitive to the timing and ordering, it is important to know when a complex event is detected (and thus an action is executed) in order to understand the behaviour of the overall system.

Historically, most of the EQL have their roots primary in active database systems [PD99], where queries are expressed by composing events using different composite operators that are inspired from the regular expressions. Examples of EQLs includes the COMPOSE language of the Ode active database [GJS92a, GJS92b], the composite event detection language of the SAMOS active database [GD94], Snoop [Cha+94] and its successor SnoopIB [AC06], GEM [MSS97], SEL [ZS01], CEDR [BCM06], ruleCore [SB05], the SASE Event Language [WDR06a], the original event specification language of XChange [BE07, Bry+07], the XSEQ [Moz+13] language for CEP over XML structured data, and the languages proposed in the following papers: [Ron98, ME01, HV02, CL04, Sán+05].

Generally, EQLs are based on composite operators including *conjunction/disjunction* of events (all/some events must happens, possibly at different times), *sequential* (all events happen in a specified order), *kleene-+* (one or more occurrence of the same types of events), and *negation* within a sequence (an event does not happen in the time between other events). Furthermore, languages such as SASE [WDR06a] also proposed the event *selection strategies* such as *skip-till-next*, *skip-till-any* and *partition-contiguity*; their details are described later.

Herein, we describe the operational semantics of CEP compositional operators using the *Snoop* event specification language [Cha+94]: Snoop is among the first contributions for defining the semantics of EQL and provides the basis for a number of CEP languages. In Snoop, an event E is a function of time-domain onto the *Boolean* values $\{True, False\}$.

$$E : T \rightarrow \{True, False\}$$

For an event of type E happening at time point τ, the function is evaluated to $True$, otherwise it is $False$. The semantics of CEP operators specified by the Snoop algebra are described as follows:

1. **Sequence Operator (;).** The sequence operator determines that two event E_1 and E_2 occur sequentially. That is, their associated timestamps are distinct to each other with one greater than the other. The formal semantics of sequential operator are as follows:

$$(E_1; E_2)(\tau) = (\exists(\tau_1)(E_1(\tau_1) \wedge E_2(\tau)) \wedge \tau_1 < \tau)$$

2. **AND/Conjunction operator (Δ).** The conjunction operator determines whether two events of type E_1 and E_2 occur at the same time. Two events are said to occur at the same time if their timestamps overlap. Formally it is defined as follows:

$$(E_1 \Delta E_2)(\tau) = ((\exists \tau_1)(E_1(\tau_1) \wedge E_2(\tau)) \vee (E_2(\tau_1) \wedge E_1(\tau)) \wedge \tau_1 = \tau)$$

3. **OR/Disjunction Operator ($\triangledown$).** The disjunction operator determines whether an event from two defined events occurs without having any constraints over the timestamps or order. Formally it is defined as follows:

$$(E_1 \triangledown E_2)(\tau) = E_1(\tau) \vee E_2(\tau)$$

4. **NOT/Negation Operator ($\neg$).** The negation operator determines the non-occurrence of certain types of events with respect to certain time interval. The non-occurrence either related to non-existence of an event or if an event does not match to the defined event attributes. It is formally defined as follow:

$$\neg(E_2)[E_1, E_3](\tau) = (\exists \tau_1)((E_1(\tau_1) \wedge E_3(\tau)) \wedge \nexists(E_2(\tau)) \wedge \tau_1 \leq \tau)$$

5. **ANY Operator.** The ANY operator returns matches if m matches of events happen out of n events in time, while ignoring the relative order of their occurrences. It is formally defined as follows:

$$Any(m, E_1, E_2, \ldots E_n)(\tau) = \exists(\tau_1, \tau_2, \ldots, \tau_{m-1})(E_i(\tau_1) \wedge E_j(\tau_2) \wedge \cdots \wedge E_k(\tau_{m-1}))$$
$$\wedge E_l(\tau) \wedge (\tau_1 \leq \tau_2 \ldots \tau_{m-1} \leq \tau) \wedge (i \neq j \neq \cdots \neq k \neq l) \wedge (1 \leq i, j, \ldots, k, l \leq n))$$

6. **Aperiodic Operator (A, A^*).** The aperiodic operator allows the expression of an aperiodic event in a time interval marked by two events. Snoop provides two different variations of the aperiodic operator: the *non-cumulative* and the *cumulative* operator. The non-cumulative aperiodic operator (A) returns matches each time an event E_2 occurs between E_1 and E_3, i.e., within the time-interval started by E_1 and ended by E_3. Formally it is defined as follows:

$$A(E_1, E_2, E_3)(\tau) = (\exists \tau_1)(\forall \tau_2)((E_1(\tau_1) \wedge E_2(\tau)) \wedge (\tau_1 \leq \tau) \wedge (+E_3(\tau_2)(\tau_1 \leq \tau_2 < \tau)))$$

The + sign indicates one or more occurrence of event E_3 after the arrival of E_2.

The cumulative aperiodic operator (A^*) returns the matches only once within the given interval of two marker events (i.e., E_1 and E_3). Formally, it is defined as follows:

$$A^*(E_1, E_2, E_3)(\tau) = (\exists \tau_1)(E_1(\tau_1) \wedge E_3(\tau)) \wedge (\tau_1 < \tau)$$

This operator accumulates the zero or more occurrence of event E_2 between events E_1 and E_3. The operator is matched with the occurrence of an event E_3.

7. **Periodic Operator (P, P^*).** Let T be a constant time $T \in \mathbb{N}^+$, then the cumulative periodic operator ($P(E_1, [T], E_2)$) detects all the occurrences of E_1 followed-by E_2 within the time T. It is formally described as follows:

$$P(E_1, [T], E_2)(\tau) = (\exists \tau_1)(\forall \tau_2)(E_1(\tau_1) \wedge + E_3(\tau_2)) \wedge (\tau_1 < \tau_2 \leq \tau) \wedge \\ \tau_1 + i \times T = \tau, \ \ for\ i \in \mathbb{N}^+\ 1 \leq i$$

The cumulative variation of the periodic operator accumulates times of occurrences of periodic events. Formally, it is defined as follows:

$$P^*(E_1, [T], E_3)(\tau) = (\exists \tau_1)(E_1(\tau_1) \wedge E_3(\tau)) \wedge \tau_1 + T \leq \tau$$

The Snoop event language has also introduced the concept of parameter contexts, which influences the detection behaviour of snoop operators. For the detection of a complex event, multiple matches might be available. Based on the semantic context of operators, different matches of primitive events are available, e.g., for the event history (a b b) during the matching of ($A;B$) pattern, the complex event might be matched once or twice depending on the semantics of the event detection system.

Based on the above algebraic operators, SASE [WDR06b, Agr+08] has provided various selection strategies for the sequence operators. Herein, we briefly described the selection strategies, namely *strict contiguity*, *skip-till-next*, *skip-till-all*. These selection strategies overload the sequence operator with the constraints and define how to select the relevant events from an input stream, while mixing relevant and irrelevant events. These selection strategies are described as follows:

1. **Strict Contiguity.** This is the most stringent event selection strategy, where two selected events within a sequence must be contiguous in the input stream. That is, given a sequence $(E_1; E_2)$, the event of type E_2 follows E_1 in a way that there can be no other events between the two selected events.

2. **Skip-till-next.** This strategy is a relaxed form of strict contiguity to remove the contiguity requirements. That is, for a given sequence $(E_1; E_2)$ all the irrelevant events between events of type E_1 and E_2 are skipped until the next relevant event is read. This strategy is important in many real-world scenarios where some events in the input are "semantic noise" to a particular pattern and should be ignored to enable the pattern matching to continue [Agr+08].

3. **Skip-till-any.** This strategy relaxes the skip-till-next by further allowing non-deterministic actions on relevant events. That is, for a given stream and a sequence $(E_1; E_2)$ all the patterns, where an event of type E_2 follows E_1, are matched and added to the output stream. This strategy essentially computes transitive closures over relevant events types as they arrive.

The more or less formalised description of the CEP operators provides an important mean to transfer knowledge about the successful design of an EQL. This allows the system designers to reuse the existing experience for building CEP solutions. In the proceeding section, we provide the details of the CEP systems that aim at providing an efficient implementation of these CEP operators.

5.3 Methods and Techniques for Complex Event Processing

The literature provides a rich set of methods and techniques utilised for efficient CEP over data streams. Existing techniques for CEP can be categorised into two classes: rule-based solutions and non-rule-based solutions. The non-rule-based approaches can further be classified into three classes: Automata-based techniques [WDR06b, Agr+08, Bre+07, BV10], Event Graph-based techniques [GJS92a, GJS92b, PD99], and Petri Nets [GFV96].

These solutions are implemented in different research prototypes or as a commercial products. In the following, we briefly review these techniques and their related implementations.

5.3.1 Rule-based Techniques

The rule-based techniques parse the defined CEP queries into a set of rules, where events are injected to a logic programming system as facts, and event patterns are specified as goals for such rules. Rule-based approaches have various advantages [Ani+10]. First, they are expressive enough and convenient to represent diverse complex event patterns, and come with a formal declarative semantics based on the well-understood logics. Moreover, declarative rules are free of side-effects (e.g. confluence problem, i.e, merger of rules). Second, the integration of CEP operators and event processing with rules is easy and natural (e.g. processing of recursive queries).

A logic-based approach introduced in [Pas06] proposes a homogeneous reaction rule language for complex event processing. It is a *combinatorial* approach for processing events and actions, with the formalisation of reaction rules in combination with other rule types, such as derivation rules, integrity constraints and transactional knowledge. Prova [Koz+06] is another rule language and a rule engine. Its design is based on reactive messaging, combination of imperative, declarative and functional programming; and it implements *SLD-algorithm* for backward reasoning. One of the important design principles in Prova is the reactive messaging that allows the organisation of several Prova rule processing engines into a network of communicating agents. A Prova agent is a rulebase that is able to send messages to other Prova agents by using message passing primitives.

One of the recent rule-based systems for stream reasoning and CEP is ETALIS [Ani+12]. It is implemented in Prolog and uses the Prolog-inference engine for event processing. It provides two event processing languages: ETALIS Language for Events (ELE) and EP-SPARQL [Ani+11]. A major distinction between ETALIS and Prova is that ETALIS is a meta-program implemented on top of a Prolog system with only one global knowledge based (KB) in which every piece of knowledge, such as incoming events, is globally applied, whereas Prova allows for local modularisation of the KB and local event processing states within the complex event computations and event message based conversations. This leads to a branching logic with local state transitions, since it is common in workflow systems and distributed parallel processing.

RETE [For90] is a rule-based algorithm and is also utilised by various CEP systems (see Chapter 4, Section 4.3.4 for details on RETE). All of the operations on data tuples like relational query processor, performing projections, selections and joins are executed on the network of RETE objects. RETE has been used by commercial CEP systems such as TIBCO Business Events [Tib] and Drools Fusion [Dro]. [SSS08] builds an EDG on top of RETE. Thus to curtain the issues of working memory and garbage collection: for condition actions (CA) (i.e., query filters), the production node from the RETE

network is solely connected to the rule node in the EDG, while for the *event condition actions* (ECA) (i.e., temporal operators) an additional event node from the EDG is connected to the rule node. The rule node fires its associated rule actions according to the ECA semantics. For an ECA rule action to fire, an event must be detected, and for its complete interval, the condition must be fulfilled. This means events are not correlated as long as RETE supplies no matched tokens.

Apart from the above mentioned strengths, event processing systems [Pas06, KS86, LLM98] based on various logic formalism have some shortcomings too. One significant shortcoming is the data or event-driven computation. Deductive systems are rather suited for a request-response computation. That is, given a request, an inference engine evaluates the available knowledge (i.e. rules and facts) and responds with an answer. This means that the event inference engine needs to check if this pattern can be deduced or not. The check is performed at the time when such a request is posed. If satisfied by the time when the request is processed, a complex event will be reported. If not, the pattern is not detected until the next time the same request is processed (though it can become satisfied in-between the two checks). Contrary to this, event processing demands data-driven computation (as handled by various approaches such as NFA , Petri Nets [GFV96], etc.), where arrival of new data evaluates the updated knowledge. Since such a process is quite frequent in a data-driven computation, deductive systems choke under high stream rates.

RETE based approaches may be integrated with deductive rules [Ani+12] to implement complex CEP operators. However, handling aggregates over event stream is a laborious and computing intensive task for the RETE-based approaches. Moreover, different event selection strategies cannot be directly implemented over RETE [Cha+94]. The main drawback of RETE-based CEP is that it has high memory space requirements. Saving the state of the system for the matched and partially matched patterns requires considerable amount of memory. The space complexity of RETE is of the order of $\mathcal{O}(RFP)$, where R is the number of rules, F is the number of asserted facts, and P is the average number of patterns per rule. Thus, if all the facts were to be compared against all the patterns, the performance of the system can degrade exponentially for a large window.

5.3.2 Graph-based Techniques

In graph-based techniques, CEP operators/rules are represented in a graph-like structure, and their execution is based on *triggers*. That is, CEP operators are triggered by the data manipulation events that occur during the processed transactions within a database or within event streams. However, the evaluation of the operators and the execution of the actions are postponed util after the commitment of the transactions.

Sentinel [Cha97, Cha+95] is an active object-oriented database that implements CEP operators defined in Snoop [AC06]. It employs an *event detection graph* (EDG) that is complied from the event expression and is directed acyclic in nature. The complex expressions are represented by the nodes in EDG with links to the nodes of their subexpressions, going all the way down to the leaf nodes of simple event definitions. The execution of the EDG starts from the bottom leaf nodes with the arrival of an event, and the execution flows upward through the graph, while satisfying/triggering defined conditions. Figure 5.2 illustrates an event detection graph in Sentinel with `AND`, `OR` operators defined over events `E1`, `E2` and `E3`. The main drawbacks of EDG, similarly to Petri Nets as described later, include the lack of support for complex operators such as kleene-+, and event selection strategies: it does not represent or even clarify the semantics of complex event expressions.

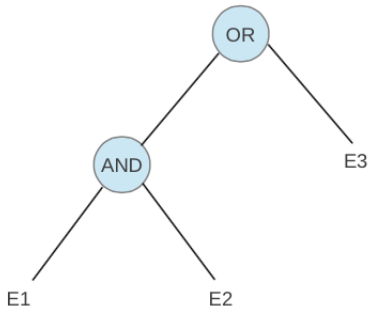

Figure 5.2: Example of Sentinel Event Detection Graph

Zstream [MM09] is another CEP systems using graph-based techniques, however, it employs a tree structure to map a set of query operators. Each tree node is assigned with a buffer, where leaf buffers (nodes) store the primitive events as they arrive. The internal nodes, containing the query operators, process events from the leaf buffers and store them in their buffers. There are two different modes of operation in Zstream, *event-based* and *batch-based*. As evident from the name, event-based process each event with its arrival, while the batch-based batches a set of primitive events and the operators are executed over them in batches.

For the detection of composite events, Gatziu *et al.* [GFV96] proposes *SAMOS Petri Nets* (S-PN): it is an extension of Coloured Petri Nets (C-PN) [Jen94]. A Petri Net [MZ95] is a collection of directed *arcs* connecting *places* and *transitions*. Places may hold *tokens* and the state or marking of a net is its assignment of tokens to places. A transition is enabled when the number of tokens in each of its input places is at least equal to the arc weight going from the place to the transition. The use of C-PN (see [Jen94] for details) allows the flow of parameter bindings through the Petri Net; thus the parameter passing within a composite event instance can be modelled. S-PN extends C-PN by allowing tokens to carry complex information regarding the parameters of events. This enables places to represent event patterns and tokens to act as the events detected up-until *now*. Furthermore, the arc expressions are used to transform the parameters of event(s) into the parameters of composite events.

Figure 5.3 illustrates the mapping and execution of conjunction, disjunction and sequence operators for events `E1` and `E2`. The simple S-PN are combined into a combination of S-PN that merge repeated patterns into a single S-PN. This enables the efficient evaluation of complex patterns. The algorithm for the execution of S-PN is based on the token-game [GFV96]. It employs a matrix to represent a set of arcs in the S-PN, and once a token (event) is added to a place, the algorithm iterates over the rows of the matrix, which represents the input-arcs-to-transitions from the place and then attempts to fire the transitions rules. If a transition is fired, the corresponding column, describing the output-arcs-to-places, is traversed and a new token is placed accordingly. The procedure is continuously repeated for each input event until no transition can be fired.

Although C-PN, employed by S-PN, provides elegant techniques to detect complex events, its executional model can become quite complex for expressive operators such as kleene-+ or large complex patterns. For instance, when several S-PNs are combined into a merged S-PN. For the same reason, the augmentation of parameters, kleene-+ operator and event selection strategies are not allowed in S-PN. Moreover, the matrix representing

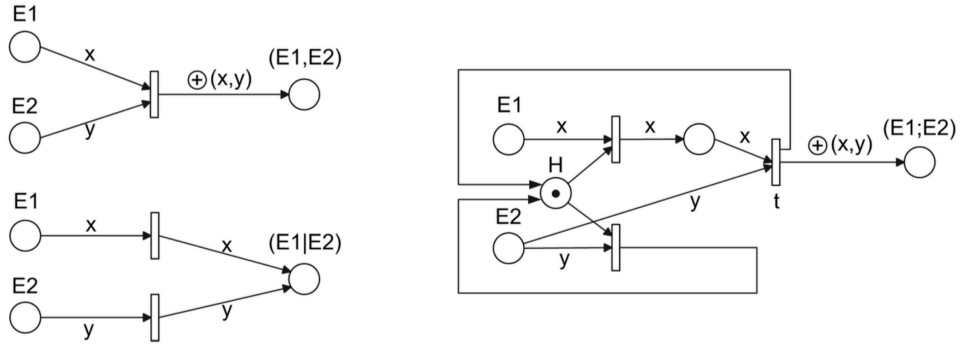

Figure 5.3: From top left to right, the S-PN of the three composite event constructors: conjunction (`E1,E2`), disjunction (`E1|E2`) and sequence (`E1;E2`). The function ⊕`(x,y)` computes the union of the parameters x and y. Note that, in the S-PN for (`E1;E2`) the place `H` (with an initial token) prevents the transition `t0` from firing until `E1` has occurred.(adapted from [MZ95])

arcs in the S-PN is usually sparse, and the algorithm has to iterate several times over rows and columns when playing token-game. Hence, this makes S-PN insufficient to provide a scalable and optimised solution. Therefore, C-PN and S-PN are only utilised for active databases, and are considered as from the very first generation of CEP systems.

5.3.3 Automata-based Techniques

Automata-based techniques are commonly used for pattern matching, where two of its variants *Non-Deterministic Finite State Automata* (NFA) or *Deterministic Finite state Automata* (DFA) are heavily utilised with customised models: CEP queries generally follow a regular expression structure, thus automata-based approaches are naturally favoured in this context. The raw event streams can be considered as an input sequence, which are matched against the defined set of states of an automaton.

Formally, a Finite State Automata (FSA) is a tuple $M(Q, \Sigma, \delta, I, F)$, where Q is a set of state, $I \in Q$ is a set of initial states, $F \subseteq Q$ is a set of final states, Σ is the alphabet, and δ is a partial mapping $\delta : Q \times (\Sigma \cup \varepsilon) \to P(Q)$ denoting the transition predicate of the states. The size of the FSA is equal to the number of states $|Q|$, and $P(Q)$ is the power set of Q. Let $q_i, q_j \in Q$, then a transition from source state to its target state $q_i \overset{P(q_i)}{\to} q_j$ for an event e happens iff e satisfies the transition predicate function $P(q_i)$. The determinism and non-determinism of a FSA, which later classifies it as DFA or NFA, depends on the transition functions. That is, if there is a same transition function for two different target states with same source state (e.g.,$q_i \overset{P(q_i)}{\to} q_j$ and $q_i \overset{P(q_i)}{\to} q_k$), then its an NFA, otherwise the automaton acts like a DFA. In general, NFA offer higher expressiveness as compared to their deterministic counterparts [Moz+13]. An NFA is able to represent complex patterns and is closed under union, intersection, conditional and kleene closure operators. The execution of an NFA automaton is realised through *runs*, which are the executional instances of an NFA automaton and represent the partial match of a sequence. Each run holds a pointer to the current active state of the FSA and a set of events that conform to its transition predicate till the current active state. A run might or might not lead to a match, thus the lifetime of a run depends on the defined window size and on the event selection strategy. For each incoming event, a new run is created if an event matches to the first state's predicate or cloned from an

existing run if the active state contains kleene-+ operator.

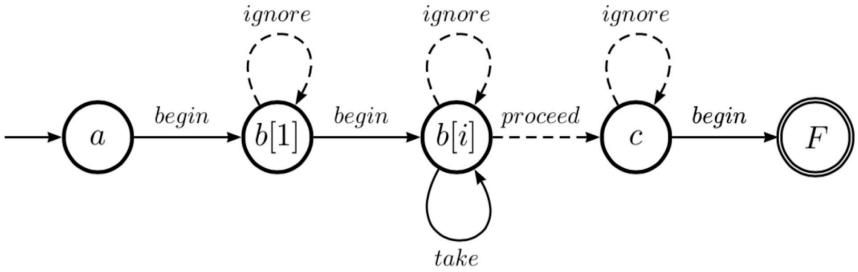

Figure 5.4: Structure of NFA^b for the pattern `ab+c` with skip-till-any strategy (adapted from [Agr+08])

SASE [WDR06b] and SASE+ [Agr+08] are based on the NFA model for the execution of CEP and provide expressible queries with event selection strategies. However, operators such as conjunction and disjunction are not supported in their system design. They extend NFA with a match buffer, called NFA^b to store the matches of each state. This allows them to utilise various run-time optimisations such as merging of runs that are stationed at the same states [Agr+08]. Figure 5.4 illustrates the NFA^b model for pattern `a,b+,c` using the skip-till-any selection strategy. The first state is labelled with the first character/predicate from the input stream, i.e., `a` with an edge labelled as *begin*. This edge transits to the next state labelled as `b[1]`, as `b` contains a kleene operator. The ignore edges describes the skip-till-next operator, and the *take* edge presents the non-determinism (ϵ). If the whole pattern matches to the input stream, it transits to the final state `F`.

Cayuga [Bre+07], which is a research project at Cornell University, provides a query language for the expression of complex event patterns called Cayuga Algebra. It also supports some special performance optimisation like indexing and garbage collector; it utilises NFA for the query compilation and execution. Esper [BV10] is a community-based CEP systems that employs a variant of DFA model. It provides an event query language with similar operator to CQL, and supports correlation and SQL-like queries over event streams. Esper also provides operators for the specification of event detection patterns and special operators to define the event stream consumption policy like the "every" operator that specifies precisely how the event stream should be matched to the pattern.

The computational complexity of the automata-based techniques depends on the number of active runs within a defined window. The higher the number of active runs, the longer it will take to process an event: each incoming event is matched with the output transitions of current active states of all the active runs. The complexity analysis of NFA is comprehensively described in [Agr+08], where each selection strategy and the kleene operator has a variable effect on the complexity measures. That is, the strict contiguity strategy has linear time complexity, skip-till-next also results in a linear time complexity, kleene-* is quadratic, and the skip-till-any is the most expensive with exponential time complexity.

The automata-based techniques provide a radically different view compared to existing DSMSs, where query operators are usually organised in an operator tree. This makes it difficult to reuse existing optimisation techniques provided by DSMSs. Such property can be considered as both an opportunity and a limitation. In one view, it can provide new exciting optimisation techniques inspired from the field of regular expression, and in another view it limits the adaptation of techniques from existing DSMSs.

5.4 Semantic Complex Event Processing

Exiting CEP systems described in Section 5.1 are constrained by the specific user-defined schemas, and primarily deal with synthetic low-level primitive events and defined actions. Thus, the integration of knowledge to extract high-level information is not taken into account. The fusion of CEP approaches with knowledge representation models, such as RDF, leads to semantic CEP (SCEP), where each event is model as an RDF triple instead of relational-tuple or XML data item. This enables SCEP systems to reap the benefits offered by the RDF data model: in particular, its schema-less nature that allows heterogeneous streams to be integrated in the system. Lifting data streams to the semantic level enables the integration of streams with higher-level knowledge representation and reasoning necessary for handling background knowledge, thus describing the context or domain in which streaming data are interpreted.

Despite the explicit advantages, the research area of SCEP is still quite fertile and only few solutions are provided in this context. The two main approaches that are related to SCEP (to some extent) are works on *temporal RDF* [GHV05] and event processing over RDF triple streams [Ani+11]. The first type of work, however, cannot be directly related to SCEP systems, but provides an intuition of time within an RDF dataset. We first review the techniques related to temporal RDF databases and then move to a discussion on event processing over RDF triple streams.

5.4.1 Temporal RDF Systems

The concept of temporal RDF [GHV05] is evolved from the annotated RDF (`aRDF`) [URS10] that builds upon annotated logics ($\mathcal{A}$) [KS92]. `aRDF` can capture fuzzy or probabilistic logics, timestamps, and temporal-fuzzy informations: an `aRDF` triple consists of an ordinary RDF triple together with an annotation, i.e., a member of $\mathcal{A}$. `aRDF` mainly emphasises on the theory, the semantics and the design of an RDF data model annotated with certain properties. Hence, in general, temporal RDF follows the path of active databases (as discussed in Section 5.3) to extract temporal relations within an RDF triple store through an extended form of SPARQL language.

In temporal RDF, generally, a time-interval/time-points can be added into RDF using a data type property. However, in order to explicitly describe the temporal query operators, temporal RDF considers time as an additional dimension in data, while preserving the semantics of the time. Thus, each triple $\langle s, p, o|\ T\rangle$ in the RDF database, called *multi-temporal RDF triple*, is associated with a timestamps $T \in \mathcal{T}$, where $\mathcal{T}$ is an n-dimensional time domain. It enables the compression of data associated with timestamps: time-stamped triples avoid the duplication of triples in the presence of temporal *pertinence* [GHV05].

Few extensions of SPARQL are proposed to query temporal RDF. T-SPARQL [Gra10] is one of them; it extends SPARQL with temporal operators, such as `OVERLAPS`, `WITHIN`. Similarly to the standard SPARQL queries and RDF triple store, T-SPARQL [Gra10] and other queries languages, such as τ-SPARQL [TB09], process the queries in an ad-hoc manner: queries are issued once and the answers are returned to the users, and data are stored in a persistent storage. Note that, both of the above mentioned works (T-SPARQL and τ-SPARQL) merely provide the theoretical details regarding the semantics of their query languages, but the implementation details and techniques to process query operators have not been the part of their work.

Temporal RDF and systems like τ-SPARQL provide the motivation of SCEP. However,

in parallel to the active databases, their aim is not to query data streams, and their executional semantics do not comply to the streaming settings. In the proceeding section, we discuss the solutions that consider RDF streams as the basic of their model and can be classified as SCEP.

5.4.2 Semantic Event Processing over RDF Streams

The semantic event processing requires a dedicated language and a framework which uniformly processes continuous queries over RDF streams. The queries provide the patterns that are matched while considering the graph nature of RDF. In this regard, to the best of our knowledge, EP-SPARQL [Ani+11] is the only system which provides a language, semantics and implementation; other systems either provide a theoretical formalism or utilise ontology-based data access, i.e., mapping the relational tuples with the domain ontology. Herein, we first review EP-SPARQL: it is directly related to our work presented in this thesis.

EP-SPARQL

It is a unified language built on top of the ETALIS [Ani+12] engine, and extends SPARQL with temporal operators. Its main building blocks are represented by a set of logical and temporal sequence operators that can be combined to express complex patterns over RDF streams. The input model of EP-SPARQL contains a set of RDF triple events, each annotated with a time interval. It is defined as follows:

Definition 5.2: RDF Event and RDF Event Stream

Let $\langle s, p, o\rangle$ be an RDF triple then an RDF event is a pair $(\langle s, p, o\rangle, T)$, where $T = [\tau, \tau']$ is an associated time-interval containing timestamps denoting the boundaries of the time interval of the occurrence. S_{re}, containing the set of RDF events, denotes the RDF event stream.

The data model of EP-SPARQL adopts two timestamps, which represent the lower and upper bound of the occurring interval, i.e., interval semantics. This reflects on output triples, whose occurrence intervals are computed from the input elements that contributed to their generation. An RDF event stream is fed to the system, where a set of sequence patterns are defined to extract the temporal and logical relationships. Herein, first we describe the EP-SPARQL query language and later provide the details about its execution model.

EP-SPARQL Query Language: The four main temporal binary operators of EP-SPARQL, which are extended from SPARQL include: `SEQ`, `EQUALS`, `OPTIONALSEQ` and `EQUALSOPTIONAL`. Query 5.1 presents a sample EP-SPARQL query with the `SEQ` operator, where the sequence for a high temperature and the detection of smoke is defined. The executional semantics of EP-SPARQL queries are based on the concept of mappings (as defined in Chapter 2), such that $(\mu, \tau_\alpha, \tau_\omega)$ is a solution for an expression of SPARQL graph pattern, and a set of them takes the form of an RDF stream, such that

$$\{(\langle s_1, p_1, o_1\rangle, \tau_1, \tau_1'), \ldots, (\langle s_1, p_1, o_1\rangle, \tau_n, \tau_n')\} \subseteq S_{re},$$
$$\tau_\alpha = min(\tau_1, \ldots . \tau_n),$$
$$\tau_\omega = max(\tau_1', \ldots, \tau_n').$$

```
SELECT ?temp ?area

            WHERE

             SEQ { ?area :hasTemp ?temp. }
             SEQ { ?area :hasSmoke ?smoke. }

             Filter (?temp > 50)
```

Query 5.1: EP-SPARQL query

In the following, we describe various operators of the EP-SPARQL language and their executional semantics. These semantics are based on sets of mappings (Ω, Ω'), compatibility ($\sim$) between mappings, and joins $\bowtie$ between mappings (see Definition 2.6 and 2.7 in Chapter 2 (Section 2.4.1)).

- `SEQ` **Operator**. The sequence operator, as described in Section 5.2.2, determines the temporal sequence between two RDF events, i.e., if an RDF event follows another. A sequence operator denoted as *SeqJoin* for the two sets of mappings is formally defined as follows

$$SeqJoin(\Omega, \Omega') = \{(\mu, \tau_\alpha, \tau_\omega) \bowtie (\mu', \tau'_\alpha, \tau'_\omega) \mid (\mu, \tau_\alpha, \tau_\omega) \in \Omega, \\ (\mu', \tau'_\alpha, \tau'_\omega) \in \Omega' \wedge \mu \sim \mu' \wedge \tau_\omega < \tau'_\alpha\}$$

- `OPTIONALSEQ` **Operator**. Similarly to SPARQL, the `OPTIONALSEQ` operator selects the RDF event within a sequence if it exists for a certain defined pattern: this operator can be classified as disjunction operator. It is denoted as *LeftJoin*, and formally, it can be defined using the left-outer join ($\bowtie$) (see Definition 2.7) as follows:

$$LeftJoin(\Omega, \Omega') = \begin{cases} (\mu, \tau_\alpha, \tau_\omega) \bowtie (\mu', \tau'_\alpha, \tau'_\omega) \mid (\mu, \tau_\alpha, \tau_\omega) \in \Omega, \\ \quad (\mu', \tau'_\alpha, \tau'_\omega) \in \Omega' \wedge \tau'_\omega < \tau_\omega : & if \ \ \mu \sim \mu' \\ (\mu, \tau_\alpha, \tau_\omega) \mid (\mu, \tau_\alpha, \tau_\omega) \in \Omega \wedge \tau'_\omega < \tau_\omega : & if \ \ \mu \nsim \mu' \end{cases}$$

- `EQUALS` **Operator**. This operator, denoted as *EqJoin*, provides the semantics of conjunction, where two RDF events are selected from the event stream if they occurs at the same time. Formally, it can be defined as follows:

$$EqJoin(\Omega, \Omega') = \{(\mu, \tau_\alpha, \tau_\omega) \bowtie (\mu', \tau'_\alpha, \tau'_\omega) \mid (\mu, \tau_\alpha, \tau_\omega) \in \Omega, \\ (\mu', \tau'_\alpha, \tau'_\omega) \in \Omega' \wedge \mu \sim \mu' \wedge \tau_\alpha = \tau'_\alpha \wedge \tau_\omega = \tau'_\omega\}$$

- `EQUALSOPTIONAL` **Operator.** This operator is the combination of both *LeftJoin* and *EqJoin*, where an RDF event is optionally selected, while considering that its time-interval overlaps with the previous ones in the sequence. It is formally defined as follows:

$$EqLeftJoin(\Omega, \Omega') = \begin{cases} (\mu, \tau_\alpha, \tau_\omega) \bowtie (\mu', \tau'_\alpha, \tau'_\omega) \mid (\mu, \tau_\alpha, \tau_\omega) \in \Omega, \\ \quad (\mu', \tau'_\alpha, \tau'_\omega) \in \Omega' \wedge \tau_\alpha = \tau'_\alpha \wedge \tau_\omega = \tau'_\omega : & if \ \ \mu \sim \mu' \\ (\mu, \tau_\alpha, \tau_\omega) \mid (\mu, \tau_\alpha, \tau_\omega) \in \Omega \wedge \tau_\alpha = \tau'_\alpha \wedge \tau_\omega = \tau'_\omega : & if \ \ \mu \nsim \mu' \end{cases}$$

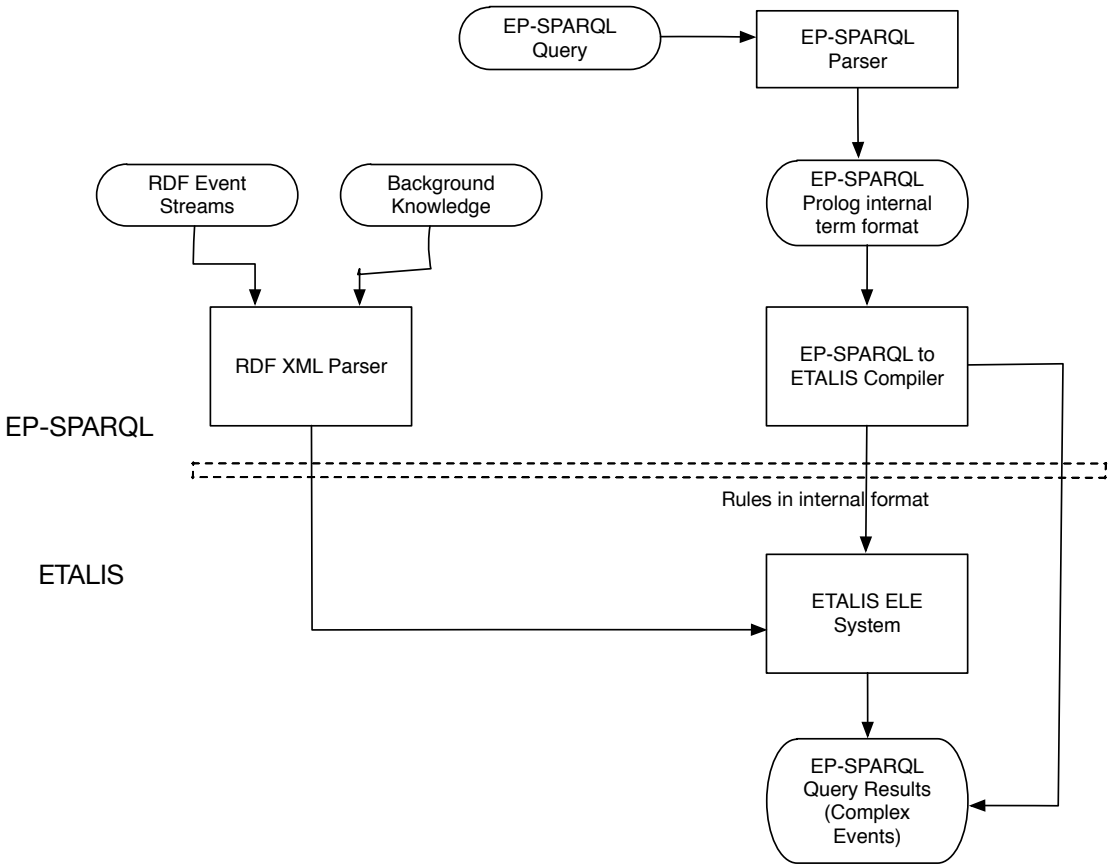

Figure 5.5: System Diagram of EP-SPARQL(adapted from [Ani+12])

EP-SPARQL does not provide any explicit negation and kleene-+ operators. However, the negation can be implemented with the combination of `OPTIONAL` and SPARQL 1.0 `FILTER` operator. Note that the semantics of the evaluation of a complete EP-SPARQL query are not provided in the literature, and cannot be inferred from the semantics of each clause: it is not clear how the nesting of the various operator can be evaluated.

Executional Model of EP-SPARQL: The executional model of EP-SPARQL is based on a rule-based system called ETALIS (as described in Section 5.3.1). The registered queries are first translated into logical expressions using the ETALIS language for events, and then to Prolog rules. ETALIS uses an event-driven backward chaining (EDBC) algorithm to compute the rules over event stream. EDBC rules are logic rules, and hence can be mixed with other rules generated from the background knowledge, i.e., domain knowledge such as an RDFS ontology.

Figure 5.5 illustrate how the EP-SPARQL queries are processed via ETALIS. The EP-SPARQL queries are translated into Prolog rules, and the RDF events within the stream are also mapped onto Prolog triples. There is also an option of using the background knowledge-base in the form of Prolog rules or an RDFS ontology with Prolog rule mappings, where the mapped RDF events are continuously matched against the translated rules using the ETALIS engine. EP-SPARQL does not implement and explicitly provide RDF-based optimisations, instead it relies on ETALIS for the execution of joins between the graph patterns and the temporal reasoning.

Other Systems

Recently, few more techniques are proposed that can be related to the SCEP approaches. The first category of these are purely theoretical with the aim of employing *ontology-based data access* to infer the relational streams through an ontology. The second approach is simply an event processing system, where pure SPARQL graph patterns (without any temporal operator) are matched with the RDF events.

STARQL [ÖMN14] (Streaming and Temporal ontology Access with a Reasoning-based Query Language) uses an ontology-based data access technique to enrich each event. It uses the first-order logic fragment for temporal reasoning over ABox sequences constructed within the query. It extends SPARQL with operators such as `USING STATIC ABOX` to query the static Knowledge-base; `SEQUENCE BY` to define sequences over triples; window operators (e.g. `NOW -> 1s`) to describe sliding windows. The data model of STARQL is a stream of Abox assertions of the form $ax\,\langle t\rangle$, where the timestamp t stems from a flow of time $(T, \leq)$ with T as a dense set of timestamps, and $\leq$ as a linear order. The defined STARQL query is mapped to the underlying SQL query that is executed over the mappings of ontology and the relational triples. STARQL provides a simple sequence operator. However, other operators, such as conjunction/disjunction, kleene-+ and negation are not supported in the language.

INSTANS [RNT12] is an event processing systems with a focus on processing an RDF event stream with standard SPARQL operators. It does not provide any CEP operators instead uses the INSERT operator in SPARQL 1.1 to insert data into SPARQL graph patterns and process them in a continuous fashion; hence it cannot be qualified as a SCEP system. It employs the RETE algorithm and propagates data through a query matching network: the matches are produced as soon as all the conditions of the SPARQL graph patterns are matched.

5.5 Summary and Discussion

How to build an efficient SCEP systems with required functionality? In this chapter, first we reviewed the existing CEP query languages and execution models, second we provided an overview of the existing SCEP systems to answer such a question. The review of the CEP systems showed that there are clear boundaries between their execution models and supported features. Table 5.1 classifies CEP and SCEP systems according to their underlying execution models and language operators. A couple of interesting points that can be inferred from our review are as follow:

1. Rule-based techniques that are inspired from the existing logic-based models can model expressive CEP operators. However, such expressibility comes with the cost of performance.

2. Non-rule-based techniques, such as NFA, that are inspired from the theory of regular expressions provide an intuitive model for matching complex temporal patterns. Although, they are not as expressive as rule-based models, they offer customised solutions for a performance intensive CEP system. These techniques are also quite efficient in effectively implementing different event selection strategies.

Hence, there is a clear trade-off between the expressibility and the performance, and the choice of a model should be carefully planned.

Table 5.1: Underlying Execution Models and Operators Supported by CEP and SCEP Systems (**S**: Sequence, **K**: Kleene-+, **C**: Conjunction, **D**: Disjunction, **EST**: Event selection strategies, **N**: Negation)

CEP and SCEP Systems	Execution Model	Supported Operators
SASE [WDR06b, Agr+08]	NFA	**S**, **K**, **EST**, **N**
ETALIS [Ani+12]	Rule-based	**S**, **C**, **D**, **EST**, **N**
SAMOS [GD94]	Petri Nets	**S**, **C**, **D**
Cayuga [Bre+07]	NFA	**S**, **C**, **D**
Esper [BV10]	NFA	**S**, **C**, **D**, **K**, **N**
Zstream [MM09]	Tree-based	**S**, **C**, **D**, **K**, **N**
Drool Fusion [Dro] & TIBCO [Tib]	RETE	**S**, **C**, **D**, **N**
EP-SPARQL [Ani+11]	Rule-based, ETALIS	**S**, **C**, **D**, **N**

Herein, based on our analysis of SCEP systems, our guide to the practitioners is as follows:

- *RDF Graph-based Event.* Following our recommendation of using the RDF graph model for RSP systems, we advocate the use of the RDF graph event model for the SCEP systems. The use of such model enables temporal reasoning over events, where each such event contains a set of triples, with each triple describing a specific attribute of the event source.
- *Take a Loan from the CEP systems.* The research area of SCEP systems is still in its early days, while CEP systems have evolved from research prototypes to commercial systems. Since the main goal of both CEP and SCEP is to reason upon the temporal properties of events, it is wise to borrow the matured executional models from CEP systems. One successful example in this case is EP-SPARQL.
- *Which Operators to use.* Early works on CEP systems, for instance Snoop [Cha+94], provide a comprehensive list of unary and binary operators, such as sequence, conjunction, disjunction, kleene+, to be utilised for temporal reasoning. This list is based on the lessons learned and real-world use cases. The second generation of CEP systems, such as SASE [Agr+08] improves this list with the inclusion of event selection strategies. Hence, this list should carefully be consulted while designing a SCEP system.

Based on the above discussion, in Chapter 8 and 9, we propose a new query language and refurbish an existing execution model for the SCEP system.

Part II

Semantically-Enabled Stream Processing: Problem Analysis, Stream Model and Proposed Solution

The basic requirement of darkness is that it enables us to extinguish the shape of an object. A girl beneath a tree, for example, with the night behind her, can only be forgotten by her absence of outline, and as the direction of darkness changed, it would reveal less and less of the tree. In this way we can select and use darkness to reveal or subdue qualities in a subject.

— Michael Donaghy

6

Problem Formulation: Continuous Query Processing over RDF Graph Streams

This chapter outlines the shortcomings of existing techniques for the RDF stream processing and provides certain clues that can be utilised for a scalable solution. We also provide an RDF graph stream model that is later utilised by our system, called SPECTRA: the details on how our system improves on the existing RSP techniques are provided in Chapter 7.

Contents

This Chapter is structured as follows: Section 6.1 provides the introductory discussion. Section 6.2 provides the insights into some of the specific limitations of the existing RDF streaming processing techniques. Section 6.3 introduces the terms and formally describes the problem of RDF graph stream processing over sliding/tumbling windows. Section 6.4 summarises the chapter.

6.1 General Idea

RDF data are modelled in terms of a set of triples $\langle subject, predicate, object \rangle$ (or $\langle s, p, o \rangle$ for short), and intrinsically form a set of labelled and directed multigraphs;

and SPARQL is the most common query language for RDF data (as described in Chapter 2). With the increase in the number of both commercial and non-commercial organisations which actively publish RDF data, the amount and diversity of openly available RDF data is rapidly growing.

As the volume of RDF data is continuously soaring, managing, indexing and querying very large collections of RDF data have become challenging. One approach to handle such large RDF data graphs is to process them using the *data stream* model [Bab+02], where streams of RDF data are processed within a predefined window. In such a model, recent elements of a stream are more important than those that arrived a long time ago: older objects are not of main concern and thus dropped. This preference for recent elements is commonly expressed using a sliding window (as described in Chapter 3, Section3.2.3), which identifies a portion of the stream that arrived between "now" and some recent time in the past. Algorithms for the so-called "streaming model" must process the incoming graphs as they arrive, while bounding a set of them within a time/count-based window. Such constraints also capture various properties that arise while processing data for dynamic domains, such as sensor networks, social networks, geospatial systems, etc., and ensure I/O efficiency when data do not fit into the main memory. RDF data in streaming environments, called *RDF graph streams*, are dynamic and updated continuously.

Supporting real-time continuous querying over RDF graph streams is challenging (NP-Complete in general settings), and achieving the level of generality requires addressing several cases rarely supported by prior works on static and RDF stream processing systems. Most notably, the ability to efficiently reuse the already computed query matches within a defined window, and adaptive and incremental indexing of the triples.

Herein, we present the rigorous analysis of the limitations and shortcomings of existing techniques for RDF stream processing.

6.2 Limitations of Existing Solutions

Analysing related work in Chapter 4 shows that certain issues recur, which we summarise here in detail.

Naively, one could approach the problem of processing RDF graph streams by leveraging existing RDF solutions [NW10b, Aba+09a, Atr+10, WKB08, Nen+15] for static data. That is, by (1) storing the entire stream, and (2) running queries for every incoming RDF graph. Clearly, this naive approach has severe limitations and is not in line with the streaming model [Bab+02]. First, storing streams obviously contradicts the idea of stream processing. Second, existing techniques utilise *offline indexing* [CN07], i.e., assuming enough workload knowledge and idle time to build the physical design before queries arrive to the system. This results in extensive indexing and expensive pre-processing of RDF graph data that may add considerable delay for each graph arriving in the stream. The third issue is the expensive recomputation of the query results under triple/graph arrival and eviction within a defined window.

Another line of solutions that may be helpful in our tasks is called *RDF stream processing* (RSP) systems [LP+11, Bar+10a, CCG10, KCF12]. These solutions comply to the streaming model, albeit in an different manner. They are based on Data Stream Management Systems (DSMSs) for un/semi-structured data streams, where each element within a stream consists of a triple, and the system has to construct the matched graphs from a set of triple streams. These solutions, as discussed later, entail expensive constraints for processing RDF graph streams, and employ *online indexing* [Sch+06] techniques.

Although, online indexing makes a step towards the more dynamic environments by allowing for continuous monitoring and periodically evaluating the index design, its performance degrades exponentially with variable workloads and increase in the number of triples within a defined window. Furthermore, most of the existing RSP systems suffer from the same problem of recomputation of the results from scratch: with the arrival or eviction of a triple, all the triples within a window are recomputed.

6.2.1 Offline/Online Indexing

Offline indexing [CN07] techniques, as used by static RDF solutions, create indices a priori assuming accurate workload knowledge and data statistics; and plenty of priori slack time to invest in physical design. But, in the context of dynamic streaming environments, such knowledge and complete dataset cannot be known a priori. Moreover, traditional indices on static RDF triples cover all triples equally, even if some triples are needed often and some never. For instance, RDF3x [NW10b] builds several clustered B+ trees for all the permutations of $\langle s, p, o \rangle$ and has a time complexity $\mathcal{O}(n)$ for index creation/update with n the number of triples. Online indexing tackles some of the above mentioned issues and is employed by RSP systems [LP+11]. The general idea is that the basic concepts of offline indices are transferred online. That is, while processing queries, the system monitors the workload and performance, it questions the need of different indices and once certain thresholds are passed, it triggers the creation of new indices and drops old ones. Such techniques perform better than offline indices in dynamic settings. However, in case of variable workloads, the creation of new indices from scratch can considerably outweigh the cost of query processing. This requires incremental indexing, where index creation and re-organisation take place automatically and incrementally.

6.2.2 Match Recomputation

The recomputation/re-evaluation of matches, once the data are updated within a window, can result in unnecessary utilisation of computation resources. Therefore, the challenge is to develop an incremental query processing, where the new query matches are computed by utilising previous query results, and the window is refreshed by only considering the effective area of the older matches. Most of the existing techniques for RSP are based on a recomputation model, i.e., with the insertion or eviction of triples in a window, query results are recomputed. As a consequence, these systems suffer from significant performance loss, as shown in our experimental study (Chapter 7 (Section 7.7)).

6.2.3 Limited Scope

Existing RSP systems are evolved from the DSMS for un/semi-structured data; hence, the use of the triple-based streaming model was an obvious choice. However, as RDF data are graphs, it is not desirable to place any limitation on the event model of RDF graph streams. The consumption of RDF graphs as triples would tear-up the joined data graph and would result in extra computation overheads for each triple update. For instance, Eddy operators [AH00] employed by CQELS, which are inherited from the relational DSMS, result in expensive computation and continuously devote resources to explore all the plans (for each input triple) and require fully pipelined execution for RDF streams. Thus, caching the statistical measure of triples and choosing a right order for every triple update causes a huge overhead (as shown in our experimental analysis (Chapter 7).

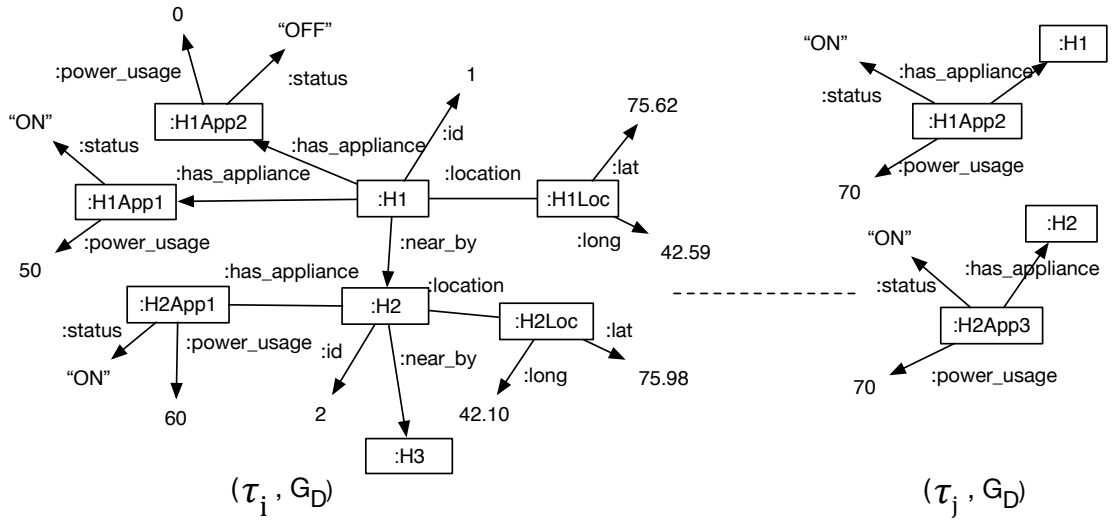

Figure 6.1: Two RDF Graph Events (τ_i, G_D) and (τ_j, G_D)

6.3 Data Model and Problem Statement

In this section, we briefly review the key concepts that form the basis of our problem definition. We also establish the notations used throughout the rest of the document.

6.3.1 Data Model

We reuse the RDF graph and SPARQL query graph definitions from Chapter 2, and based on this, we define an RDF graph event and RDF graph streams as follows.

Definition 6.1: RDF Graph Event

*An **RDF graph event** denoted as (τ_i, G_D), consists of an RDF data graph G_D and a timestamp $\tau_i \in TS$, where TS is a set of totally ordered timestamps.*

Definition 6.2: RDF Graph Stream

*An **RDF graph stream**, denoted as $\mathcal{S}_g$, is a possibly infinite set of RDF graph events.*

A conjunctive SPARQL query graph is used to match an RDF graph event within the RDF graph stream, and is represented by a set of triple patterns. A triple pattern *tp* is an RDF triple, where query variables (*vars*) are allowed in any position. The set of such triple patterns is called a basic graph pattern: we denote it as query graph G_Q. Triple patterns are usually connected by the shared subjects or objects and a join occurs on their shared subjects or objects. In this chapter, we only consider connected query graphs and we do not consider predicate joins, because variable predicates are not very common as shown in the previous study [AF11]; hence predicates of triple patterns are constants ($p \notin vars$).

An example SPARQL query, which retrieves all the appliances – with the *status* `ON` – of a house located *nearby* house with id `2` is expressed in Query 6.1.

```
SELECT ?house, ?app, ?nbhouse
  WHERE {

      ?house <has_appliance> ?app.
      ?app <status> ''ON''.
      ?house <near_by> ?nbhouse.
      ?nbhouse <id> 2.

        }
```

Query 6.1: Illustrative SPARQL Query for the Smart Grid Use case

Processing a query graph G_Q against an RDF data graph G_D amounts to finding all the subgraph isomorphisms (subsequently homomorphism) between G_Q and G_D. The result of a *select* query graph, however, is not itself a graph but – in analogy to SQL – a set of rows, each containing a distinct set of bindings of query variables in $\mathcal{V}$ to constants.

We now describe the sliding windows that are used to extract a specific set of recent RDF graph events.

Definition 6.3: Sliding Window

*A **sliding window** $R_W(\tau) = W_x^\omega(\mathcal{S}_g)$, which contains the slide x and window size ω ($x, \omega \in \mathbb{N}^+$), at each time τ converts the stream $\mathcal{S}_g$ into a set R_W containing recent RDF graphs from the stream $\mathcal{S}_g$, such that*

$$R_W(\tau) = \{G_D \mid (\tau', G_D) \in \mathcal{S}_g \wedge \tau_b \leq \tau' \leq \tau_e\},$$

where the window at time τ begins at $\tau_b = \lfloor \frac{\tau - \omega}{x} \rfloor \times x$ and ends at $\tau_e = \tau_b + \omega$.

Windows are a central concept in stream processing: an application cannot store an infinite stream in its entirety. Instead windows are used to summarise the most recent set of elements [KS09b] and evict the older ones from the system. For example `Range 5 Hours Slide 10 Minutes` describes a window of size `5 Hours` and a slide `10 Minutes` determines the granularity at which the window borders change.

Now we reuse the concept of SPARQL mappings (μ) (from Chapter 2, see Definitions 2.7 and 2.8) to define the evaluation function over the RDF graph event and the defined window.

Definition 6.4: Evaluation of $[\![P]\!]_{G_D}$

If P is a SPARQL graph pattern, μ is a partial function $\mu : \mathcal{V} \to \mathcal{B} \cup \mathcal{L} \cup \mathcal{I}$, and G_D is an RDF graph, then its evaluation is described as follow:

$$[\![P]\!]_{G_D} = \{\ \mu \ \mid\ dom(\mu) = vars(P)\ \wedge\ \mu(P) \subseteq G_D\}$$

Based on the above definition, we describe the algebraic properties of P over G_D as follows. We reuse the concept of mapping (μ), compatibility between mappings ($\sim$), selection (π), and union of mappings ($\cup$) from Chapter 2.

Definition 6.5: Evaluation of $[\![P]\!]_{G_D}$

Let G_D be an RDF graph, tp is a triple pattern, P, P_1 and P_2 are SPARQL graph patterns, R is a filter condition, and $v \subset \mathcal{V}$ a set of variables. The semantics of SPARQL graph patterns over an RDF graph are recursively defined as follows:

- $[\![tp]\!]_{G_D} := \{\mu \mid dom(\mu) = vars(tp) \ \wedge \ \mu(tp) \in G_D\}$
- $[\![P_1 \ \bowtie \ P_2]\!]_{G_D} := \{\mu_1 \cup \mu_2 \mid \exists \mu_1 \in [\![P_1]\!]_{G_D} \wedge \exists \mu_2 \in [\![P_2]\!]_{G_D} \wedge \mu_1 \sim \mu_2\}$
- $[\![P_1 \ \texttt{AND} \ P_2]\!]_{G_D} := [\![P_1]\!]_{G_D} \bowtie [\![P_2]\!]_{G_D}$
- $[\![P_1 \ \texttt{UNION} \ P_2]\!]_{G_D} := [\![P_1]\!]_{G_D} \cup [\![P_2]\!]_{G_D}$
- $[\![P_1 \ \texttt{OPT} \ P_2]\!]_{G_D} := [\![P_1 \ \bowtie \ P_2]\!]_{G_D} \cup \{\mu_1 \in [\![P_1]\!]_{G_D} \mid \forall \mu_2 \in [\![P_2]\!]_{G_D}, \mu_1 \nsim \mu_2\}$
- $[\![P \ \texttt{FILTER} \ R]\!]_{G_D} := \{\mu \in [\![P]\!]_{G_D} \ \mid \ \mu \models R\}$
- $[\![\ \texttt{SELECT}_v(P)]\!]_{G_D} := \pi_v([\![P]\!]_{G_D})$

The evaluation of a SPARQL graph pattern P over an RDF graph event, and stream is defined as follows.

Definition 6.6: Evaluation of $[\![P]\!]$ over RDF Graph Event and Stream

Let (τ, G_D) is an RDF graph event, $P \in G_Q$ is a graph pattern, and $\mathcal{S}_g$ is an RDF graph stream, then evaluation of P over them is defined as follows:

$$[\![P]\!]_{(\tau, G_D)} = (\tau, [\![P]\!]_{G_D})$$

$$[\![P]\!]_{\mathcal{S}_g} = \{(\tau, [\![P]\!]_{G_D}) \mid (\tau, G_D) \in \mathcal{S}_g\}$$

Example 5 *Recall Query 6.1 (G_Q) and an RDF graph event (τ_i, G_D) in Figure 6.1. Then evaluation of Query 6.1 over such an event will result in the following set of mappings associated with the timestamp τ_i.*

$$[\![G_Q]\!]_{(\tau_i, G_D)} = (\tau_i, \{ \ \texttt{(:H1, :H1App1), (:H1App1, 'ON'),} \\ \texttt{(:H1, :H2), (:H2, '2').}\})$$

Recall that a defined window collects a set of RDF graphs within its boundaries. Thus, we abuse the evaluation function $[\![.]\!]_{(\tau_i, G_D)}$ and extend it for window $(R_W(\tau))$ noting $[\![tp]\!]_{R_W(\tau)}$ to present our problem statement.

6.3.2 Problem Statement

Based on the above concepts, we formally describe our problem statement as follows.

Problem 1 ***Given*** *an RDF graph stream $\mathcal{S}_g$, a window $W_x^{\omega}(\mathcal{S}_g)$ over such stream, and a query graph G_Q, we want to* ***evaluate*** *$[\![G_Q]\!]_{(\tau_i, G_D)}$ i.e., execute a query graph G_Q over an event $(\tau_i, G_D) \in \mathcal{S}_g$ at τ_i, such that*

$$[\![G_Q]\!]_{(\tau_i, G_D)} \ \oplus (Op) \ [\![G_Q]\!]_{R_W(\tau_j)} = [\![G_Q]\!]_{R_W(\tau_i)},$$

where $\tau_j < \tau_i$, and operator $\oplus$ incrementally uses the previously evaluated mappings within a window, and $Op \in \{\bowtie, \texttt{AND, OPT, UNION}\}$.

Without loss of generality, we only consider time-based windows. Other flavours, such as count-based windows, can easily be integrated into our model. Herein, while utilising the complexity analysis of SPARQL, we provide an overview of the complexity of Problem 1.

Theorem 6.1

The general complexity of processing an RDF graph event (τ_i, G_D) against a query graph G_Q can be described as $\mathcal{O}(|G_Q| \cdot |G_D|)$, while for a window $(R_W(\tau_i))$ at time τ_i, it is solved in $\mathcal{O}(|G_Q| \cdot |R_W(\tau_i)|)$.

Proof Sketch. Theorem 6.1 can easily be extended from the Theorem 2.2. □

The size of each RDF graph G_D within an event, which is bounded by the defined window, has a huge impact on the query performance. As noted in Section 6.2, existing solutions employ the index-store-query model to process RDF streams or querying RDF data: this may not be a viable solution for processing RDF graph streams. Thus, our first goal is to prune each RDF graph event according to the defined query graph. This consequently reduces the search space before the execution of the query graph for each event. In Chapter 7, we provide a graph pruning approach to extract a summary graph from each RDF graph event. This removes all the unnecessary triples from each event and aims at limiting the search space for joining the mappings of triple patterns within a query graph.

6.4 Summary

In this chapter, we outlined the main issues and limitations of RSP and static RDF graph processing systems. We also provided a new data model for RDF streams called RDF graph streams, where a set of triples can be part of the RDF graph event. Based on this, we discussed the problem statement of the RDF graph stream processing. Our contributions for this chapter are as follow.

- **Limitations of Existing Systems.** We detailed the limitations of existing RSP and static RDF graph processing systems.
- **RDF Graph Event and Streams.** We presented a new event and streaming model for RDF graph streams.
- **Problem Formulation.** We provided the formal problem statement for processing RDF graph streams.

According to our observations, existing RSP systems are directly inspired from the traditional DSMSs, and existing static RDF solutions are too costly to be integrated in the streaming setting. Therefore, the multigraph nature of RDF demands customised optimisation in streaming settings. These optimisations, such as incremental indexing and incremental evaluation of streams, can lead to a system that can be compared with DSMSs in terms of performance and scalability. In the following chapter, while considering these observations, we provide the details of our RDF graph stream processing system called SPECTRA.

Quand tu veux construire un bateau, ne commence pas par rassembler du bois, couper des planches et distribuer du travail, mais réveille au sein des hommes le désir de la mer, grande et large.

If you want to build a ship, don't drum up the men to gather wood, divide the work, and give orders. Instead, teach them to yearn for the vast and endless sea.

— Antoine de Saint-Exupéry

7

SPECTRA: High-Performance RDF Graph Streams Processing

In this chapter, we provide the initial contributions necessary for a scalable and optimised solution for RDF graph stream processing. In particular, we provide the design of an incremental and adaptive indexing technique that is optimised for frequent updates. Subsequently, we detail the underlying data structures and incremental query matching algorithms: our framework called SPECTRA that incorporates such functionalities is explained in this chapter. We reuse some of these techniques in Chapter 8 for the integration of Semantic Complex Event Processing.

Contents

This chapter is structured as follows: Section 7.2 presents an overview of the SPECTRA framework and its main operators. Section 7.3 presents the SummaryGraph *operator of SPECTRA that prunes the unwanted triples from each RDF graph event. Section 7.4 provides the details of the incremental*

indexing and how to continuously process query graphs. Section 7.5 details the incremental evaluation of the RDF graph streams. Section 7.7 presents the empirical evaluation of SPECTRA and its comparison with existing techniques. Section 7.8 illustrates some of the extensions that can be built on top of SPECTRA. Section 7.9 concludes the chapter.

7.1 Introduction

This chapter introduces SPECTRA, an in-memory framework that tackles the challenge of continuously processing RDF graph streams in an incremental manner. As a framework, SPECTRA combines RDF graph summarisation and an efficient data structure – called *query conductor* – with an incremental and adaptive indexing technique to match a set of RDF graphs within a sliding window. To avoid storing and processing all the graph objects from the streams, we exploit the structure of the query to prune irrelevant information. That is, the registered query is used to prune all the triples that do not match the subjects, predicates and objects of the patterns defined in the query. The pruned set of triples, called *summary graph*, is used to implement multi-way joins between the set of triple patterns in the query graph. This results in pruning all the invalid triples without incurring storage and query processing costs.

Summarised RDF graph events are materialised into a set of vertically partitioned [Aba+09a] *views*, where each view – a two-column table (s, o) – stores all the information for each unique predicate in the summarised RDF graph. Here we use our incremental indexing technique inspired from the database cracking [Idr+11, IKM07] to index the joined (s, o) pairs within the set of views. It is a fully dynamic approach as it assumes no workload knowledge and requires no idle time for its creation: indices are built continuously, partially and incrementally as part of the query processing. A set of views represents the universe of the triples to be matched, and hash-join operations between views are used to join the triples based on subject/object column. The joined triples are incrementally indexed using a sibling relationship between them, enabling SPECTRA to support complex queries. A *timelist* is also used to associate the indexed triples with their respective timestamps, which permits the system to detect the older matches, as the window slides. Our experimental analysis confirms the eminence of our methods and shows that SPECTRA outperforms state-of-the-art solutions up to an order of magnitude.

7.2 Overview of the SPECTRA Framework

For the incremental evaluation of events, the essence is to keep track of the previously matched results, and compute the new matches by only considering the effective area of the previously stored matches. Similarly, when the window slides, it should evict the deceased matches and propagate all the matches that are no longer valid. The SPECTRA framework directly maintains a set of vertically partitioned tables, called *views*, that contain the up-to-date matches. Through incrementally produced indices for matches associated with their timestamps, our solution can handle both the insertion and eviction of matches without computing all the triples from scratch within a window.

Algorithm 1 illustrates the global execution of the SPECTRA query processing. Each incoming event from a stream is first subjected to the graph summarisation process (*line 2*) (Section 7.3), where the query structure is utilised to prune "dangling" triples from each event before starting the matching process. The summarised events are materialised

Algorithm 1 SPECTRA query processing: main process

```
1: for each event (τ_i, G_D) ∈ S_g do
2:     views ← GraphSummary(G_D, G_Q)                                  ▷ Section 7.3
3:     eventMatches ← QueryProc(views, G_Q, τ_i)                       ▷ Section 7.4
4:     if completeMatch(eventMatches, G_Q) then
5:         prevMatches ← eventMatches
6:     else
7:         prevMatches ← IncQueryProc(prevMatches, eventMatches, G_Q, τ_i)   ▷ Section 7.5
8:     end if
9: end for
```

into a set of views using *bidirectional multimaps* to enable fast hash-joins. Then the system implements the joins between the views according to the triple patterns defined in the query graph (*line 3*). It incrementally constructs the indices between the joined triples as a by-product of the join process. If this process results into complete matches for a query graph, the timestamp of the event and incrementally produced indices are used to tag the matches in a timelist; and new matches are persisted (Section 7.4). Otherwise, the partial matches produced during the initial join process are processed with the previously computed/persisted matches (*line 7*), while employing the incremental indexing. If this results into new matches, they are also persisted to be utilised later. This process ensures the completeness of the matches for a query graph for all the different types and sizes of events within a window (Section 7.5).

7.3 RDF Graph Summarisation

Graph summarisation is the process of summarising a graph into a smaller graph that retains the useful characteristics of the original RDF data graph. That is, ignoring the part of the graph that contains no relevant triples with respect to the query. Thus, the query processing can be faster on summarised graphs than on the un-pruned ones [Gur+14]. Some RDF graph stores have extended bi-simulation and locality-based clustering approaches to perform join-ahead pruning via graph summarisation [Pic+12, Zou+11]. *Bi-simulation-based summaries* [Pic+12] are effective if only predicates are labelled with constants, such that multiple possible disconnected components of data graphs are merged into compact synopsis for indexing. It is an approximate solution and may contain errors. *Locality-based summaries* [Zou+11] use essentially graph clustering in which vertices of a data graph are partitioned such that vertices within each partition share more neighbours than nodes that are spread across the partitioning. Locality-based approaches are particularly effective if one or more of the subjects or objects in the query graph are labelled with constants.

Both of the above mentioned approaches are effective in static settings, where (i) data pre-processing delays are not of main concern, (ii) the complete dataset is known a priori for statistical analysis, and queries are unknown; therefore, data must be stored and indexed such that any possible kind of query can be answered efficiently. However, this is not the case in stream processing environments: queries are processed continuously, and the complete dataset is not available beforehand for the analysis. This provides various interesting opportunities to get precise summaries of data graphs. It includes: (i) events can contain a large number of triples but the query graphs usually touch arbitrary small parts of the events; (ii) as query graphs are known in advance, they can be treated

as an advice of how data should be stored as summary graphs to process them. These properties are utilised by our query-based graph summarisation technique.

Observation 1 *Given a query graph G_Q and an event (τ_i, G_D), the number of triple patterns $|tp| \in G_Q$ is less than or equal to the number of triples $|t| \in G_D$.*

In general, query graphs are focused on a specific part of the graph to be matched. Thus pruning the unnecessary triples that would not be utilised during the query processing would greatly reduce the search space. Furthermore, query graphs typically involve in finding the connected components of the input graph G_D. Thus, if one or more subjects or objects are labelled with constant, we can safely prune all the false positives from the results. Based on the above observation, we define the query-based RDF summary graph as follows.

Definition 7.1: Query-based RDF Summary Graph

*A **query-based RDF summary graph** G_S for a given query graph G_Q and an RDF graph $G_D \in (\tau_i, G_D)$ is an RDF graph, where a triple $t \in G_D$ is in G_S iff,*

- $\exists \texttt{pred}(t)$, *such that,* $\texttt{pred}(tp) = \texttt{pred}(t)$, *where* $tp \in G_Q$
- *and if* $\texttt{subj}(tp) \notin \mathcal{V}$, $\texttt{subj}(tp) = \texttt{subj}(t)$, *and if* $\texttt{obj}(tp) \notin \mathcal{V}$, $\texttt{obj}(tp) = \texttt{obj}(t)$.

The function `subj`, `pred`, *and* `obj`, *extracts the subject, predicate and object values from triple and triple patterns respectively.*

The set of triples within each summary graph G_S is stored using vertically partitioned tables, called *views*, each denoted as $\mathcal{V}_j$ with j a predicate label. An example is shown in Figure 7.1(a): it is a summary graph produced using Query 1 for the RDF graph event G_D^i (in Figure 7.1(a)). Figure 7.1(b) shows a set of views, each containing a set of (s, o) pairs for a unique predicate, for the summary graph G_S^i.

Example 6 *Consider* Query 1, *it contains four distinct predicates* (`<has_appliance>`, `<status>`, `<near_by>`, `<id>`) *and therefore only triples with those predicates are required for query processing. Thus, all the RDF triples in G_D of RDF graph event (τ_i, G_D) (see Figure 6.1), which are not associated with those four predicates can safely be pruned, without introducing false negatives to the results. Furthermore, there are two constants (*`ON` and `2`*) at the object level in* Query 1. *Both of these constant objects can also be utilised to further reduce the size of the summary graph (see Figure 7.1(a)). One can also notice that this form of join-ahead pruning allows us to detect empty join results without even starting the pattern matching for an event.*

The execution of the GraphSummary operator is shown in Algorithm 2. It takes the input RDF graph of an event (τ_i, G_D), the registered query graph G_Q and a set of views, each for a $tp \in G_Q$. The algorithm performs the pruning and vertical partitioning of triples $t \in G_D$. First, we compare the predicate values for each triple pattern $tp \in G_Q$ to the predicates of triples $t \in G_D$ (*line 5*). Second, if the subject or/and object of the triple pattern contains a constant, we also compare it with the subjects/objects of triples in $t \in G_D$ (*line 6*). Finally, the matched set of triples for each triple pattern is encoded and stored in the corresponding view (*line 7*). By encoding, we mean encoding strings to numeric values by using a dictionary [Baz+15], a routine process in RDF storage

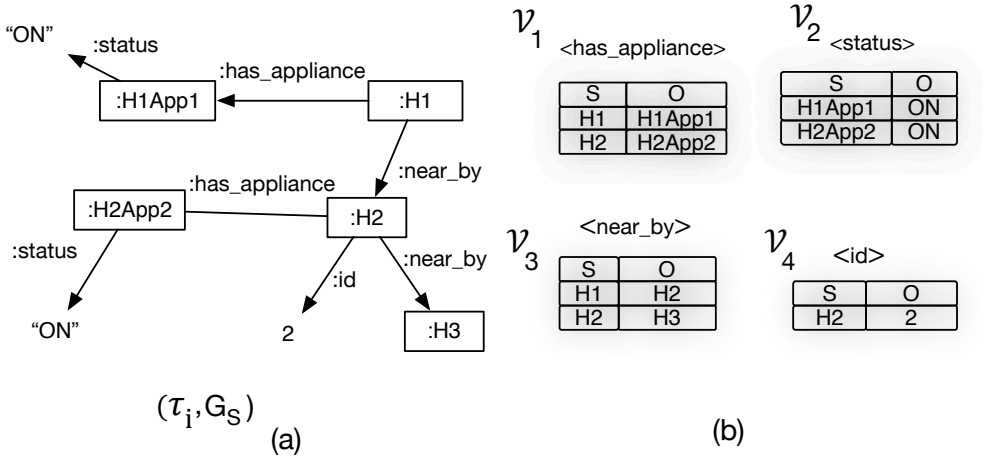

Figure 7.1: (a) Summary Graph from the RDF Graph Event (τ_i, G_D) using Query 6.1, (b) Materialised Views for the Summary Graph (τ_i, G_S).

Algorithm 2 RDF Graph Summarisation

1: $viewset \leftarrow \left\{\mathcal{V}_{tp_1}, \mathcal{V}_{tp_2}, \ldots, \mathcal{V}_{tp_{|G_Q|}}\right\}$
2: **procedure** GRAPHSUMMARY (G_Q, G_D)
3: **for each** triple pattern $tp \in G_Q$ **do**
4: | **for each** triple $t \in G_D$ **do**
5: | | **if** `pred(`tp`)` $=$ `pred(`t`)` **then**
6: | | | **if** (`subj(`tp`)` $\in vars$ **or** (`subj(`tp`)` $\notin vars$ **and** `subj(`tp`)` $=$ `subj(`t`)`) **and** (`obj(`tp`)` $\in vars$ **or** (`obj(`tp`)` $\notin vars$ **and** `obj(`tp`)` $=$ `obj(`t`)`))) **then**
7: | | | | $\mathcal{V}_{tp} \leftarrow \mathcal{V}_{tp} \cup \{\text{ENCODE}(t)\}$
8: | | | **end if**
9: | | **end if**
10: | **end for**
11: **end for**

systems. This greatly compacts the dataset representation and increases performance by performing arithmetic comparison instead of string comparison. Furthermore, dictionary encoding also caters the blank nodes and allows the matching process of each event in a manner consistence with the RDF data model [Hog+14].

7.4 Continuous Query Processing

Here, we first provide the basis of our incremental indexing technique and the data structure called query conductor, and then we describe the details of the query processing (QUERYPROC) operator.

7.4.1 Incremental Indexing

The computation of *s-s* joins between a set of views is a straight-forward procedure, and can be realised without the use of any indexing. However, complications arise when there are *s-o* / *o-s* joins between a set of views, i.e., for cyclic or tree-structured query graph patterns. Furthermore, for incremental pattern matching, the system requires to locate the correct part of the matches that are affected with the arrival of new events or with

the eviction of old ones when the window slides. Generally, in static settings (as well as in RSP) indices based on B+tree are used to locate the (s, o) pairs for multi-way join operations. These indices are discarded or built-up from scratch with new updates, thus incurring delays during the rebuilding process.

To achieve both high performance and scalability, our index creation and maintenance solution is a by-product of join executions between views. Given a set of disjoint views, only the joined triples are indexed using *sibling lists*; each for a unique view. Each sibling list is composed of *sibling tuples* as defined below.

Definition 7.2: Sibling Tuple

*A **sibling tuple** is a 3-tuple* $st = (\texttt{id}(v), \texttt{id}(u), g_i)$*, where* v *and* u *are the* (s, o) *pairs joined on subject/objects between views,* `id` *is a function which assigns monotonically increasing numeric values to* (s, o) *pairs, called pair-ids. Finally,* g_i *is an ordinal number, which is assigned increasingly monotonically for each unique graph within the set of joined* (s, o) *pairs.*

Given a join between two views, triples that will match during the join operation are *siblings*. Siblings with the same values for the join attributes are given the same ordinal number (g_i). Two sibling tuples st_1 and st_2 belong to the same matched graph if $g_1 = g_2$, where $g_1 \in st_1$ and $g_2 \in st_2$. The set of sibling tuples incrementally represents the structural relationship, i.e., multiple matched graphs, between the joined (s, o) pairs within views.

In our current implementation, we use a flat list as an underlying data structure for each *sibling list*, such that the pair-ids of sibling tuples are placed at $3i$, $3i + 1$, while an ordinal number is placed at $3i + 2$. All the sibling tuples that are before $3i$ are built earlier and all the sibling tuples that are after $3i + 2$ are built later. This information can be used to speed up the join process, i.e., the (s, o) pairs can be answered at the cost of searching the sibling list, only. Moreover, if there are no matches to be found, the sibling list significantly restricts the values of the triples that have to be analysed by the query.

The set of sibling tuples conceptually describes an undirected graph between the joined (s, o) pairs, which as a result enables an efficient strategy to cater cyclic queries. That is, if one computes the query $tp_1(x, p1, y) \bowtie tp_2(y, p2, z) \bowtie tp_3(z, p3, x)$, which lists all the triangles within the summary graphs, as a sequence of two join operations: sibling relationships between the joined (s, o) pairs are utilised to check if a set of sibling tuples associated to the same ordinal number satisfies the cyclic relationships. § We utilise a set of sibling lists, each for a view, to enable the dynamic reordering of join operations, as described later. Note that the pair-ids and ordinal numbers are iteratively produced during the join operation, thus a sibling list remains sorted throughout its lifetime.

Example 7 *Consider* $\mathcal{V}_1$ *and* $\mathcal{V}_2$ *in Figure 7.2(a). The object column of* $\mathcal{V}_1$ *is joined with the subject column of* $\mathcal{V}_2$*, each distinct* (s, o) *pair within both views is assigned with a pair-id (Figure 7.2(a)). The* (s, o) *pair* `(H1,H1App)` *identified with an id* `1` *in the view* $\mathcal{V}_1$ *has two joined* (s, o) *pairs in* $\mathcal{V}_2$*:* `(H1App,H1App1)` *and* `(H1App, H1App2)` *with ids* `4` *and* `5` *respectively. Thus, two sibling tuples* st_1 *and* st_2 *with the associated ordinal number* `00` *(only two bits ordinal numbers are used for the sake of presentation, we use 64 bits numbers for the implementation purposes) can be constructed as* `(4,1,00)` *and* `(5,1,00)` *(Figure 7.2(b)). Similarly, join between the* (s, o) *pairs identified with* `2` *and* `6` *is indexed with sibling tuple* st_3 `(6,2,01)`*; and the join between* (s, o) *pairs identified with*

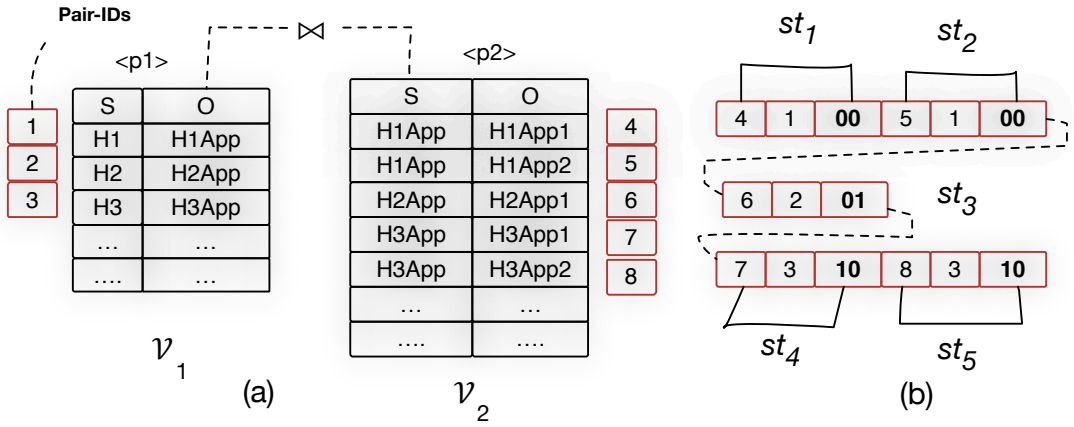

Figure 7.2: (a) Two Views Joined on an Object and Subject Column, (b) Sibling List constructed during the Join Operation for $\mathcal{V}_2$

3, 7 and 8 is indexed with sibling tuples st_4 and st_5. The final matched result of such join operation is extracted by first collecting the distinct sibling tuples according to the ordinal numbers and then collecting all the (s, o) pairs associated to a distinct ordinal number. This procedure is described in the next section.

Query Conductor

We now introduce the underlying data structure, called *query conductor*, used for the SPECTRA framework.

Definition 7.3: Query Conductor

*A **query conductor** is a data structure that first stores the materialised views from the summary graph (G_S) and then stores the joined (s, o) pairs evaluated from processing a query graph (G_Q). It consists of three components:*

1. *a set of bidirectional multimaps, each stores the (s, o) pairs associated to a predicate (i.e., views),*
2. *a set of sibling lists to identify the sibling relationships between joined (s, o) pairs (sibling tuples),*
3. *a timelist to efficiently detect the obsolete (s, o) pairs with the slide of the query window.*

The design of the query conductor is motivated by the fact that we require a data structure that not only is suitable for write-intensive operations, but also provides fast joins between views, while considering the temporal properties. The components of a query conductor are utilised in multiple different configurations during different steps of the algorithmic operations, and are briefly described below:

1. Multimaps are containers that associate values to the keys in a way that there is no limit on the number of same keys. It provides constant look-ups (considering there

are no hash-collisions) for the keys (typically for *s-s* joins between two views). In order to provide constant look-ups for objects-related joins (i.e., *s-o*, *o-s*, *o-o*), by join operator reordering between two views, we extend the multimaps to bidirectional multimaps. That is, there is no limit on the number of same keys, and it also provides constant look-ups for the values.

2. The sibling lists are used to implement the incremental indexing of matched (s, o) pairs in a set of views; it also assists in detecting the dropped (s, o) pairs with the slide of a window.

3. The timelist stores the monotonically increasing timestamps of the events and provides a flat structure, since tree-like structures cannot cope with the frequent object insertion and deletion.

We illustrate various components of our query conductor by an example given below.

Example 8 *Recall* QUERY 6.1 *(Section 6.3) that consists of four triple patterns with three join operations. Views* $\mathcal{V}_1, \mathcal{V}_2, \mathcal{V}_3$ *and* $\mathcal{V}_4$ *in Figure 7.3(a) represent respectively the materialised triples from a summary graph for each triple pattern. The join operations and the construction of sibling lists* SL_1*,* SL_2*,* SL_3 *and* SL_4*, each for a view, are illustrated in Figure 7.3 (a). First, we use* $\mathcal{V}_1$ *and* $\mathcal{V}_2$ *to implement the s-o join* ① *between* tp_1 *and* tp_2*; such join produces two intermediate result sets, called result-views* $\mathcal{RV}_1$ *and* $\mathcal{RV}_2$*; each with a set of* (s, o) *pairs. The sibling lists* SL_1 *and* SL_2 *are populated incrementally by building the sibling tuples using the pair-ids, as shown in Figure 7.3 4 (a)* ①*. Next,* $\mathcal{RV}_1$ *and* $\mathcal{V}_3$ *are used to implement the s-s join* ② *between* tp_1 *and* tp_3*. The hash-join produces* $\mathcal{RV}_3$*, and using the ordinal numbers and pair-ids in* SL_1*,* SL_3 *is constructed, as shown in Figure 7.3 (a)* ②*. Finally, using* $\mathcal{RV}_3$ *and* $\mathcal{V}_4$*, the o-s join* ③ *between* tp_3 *and* tp_4 *results in only one* (s, o) *pair in* $\mathcal{RV}_4$ *with ordinal number* `00` *in* SL_4*. In order to retrieve the final set of matched* (s, o) *pairs, we first select the resulted view with the smallest size (i.e.,* $\mathcal{RV}_4$*) and using its associative sibling list (*SL_4*), we extract the valid* (s, o) *pairs from all the other resulted views conforming to the sibling tuples. This approach is similar to depth-first-search (DFS). Figure 7.3(b) shows the set of final views* $\mathcal{FV}$ *of* (s, o) *pairs with their respective predicates. The ordinal number/numbers associated with the resulted* (s, o) *pairs in sibling tuples are also assigned to a timestamp in the timelist (see Figure 7.3(b)).*

7.4.2 Query Processor

In this section, we provide the details of the query processor (QUERYPROC) operator from Algorithm 1 (*lines 3-5*). Its main objectives are as follows:

1. match an event with the query graph, if the process produces fully matched subgraphs then proceed towards the eviction process,

2. if the process produces only partially matched subgraphs within an event, then proceed towards the incremental query processor (Section 7.5).

QUERYPROC, as described in Algorithm 3, takes seven elements as input: a query graph G_Q, the set of views for an event obtained from GRAPHSUMMARY operation, a set of final views ($\mathcal{FV}$) that contains complete matched results up to this point in a window, the timestamp of the current event τ_i, and the timelist τ_L with the window size ω and slide

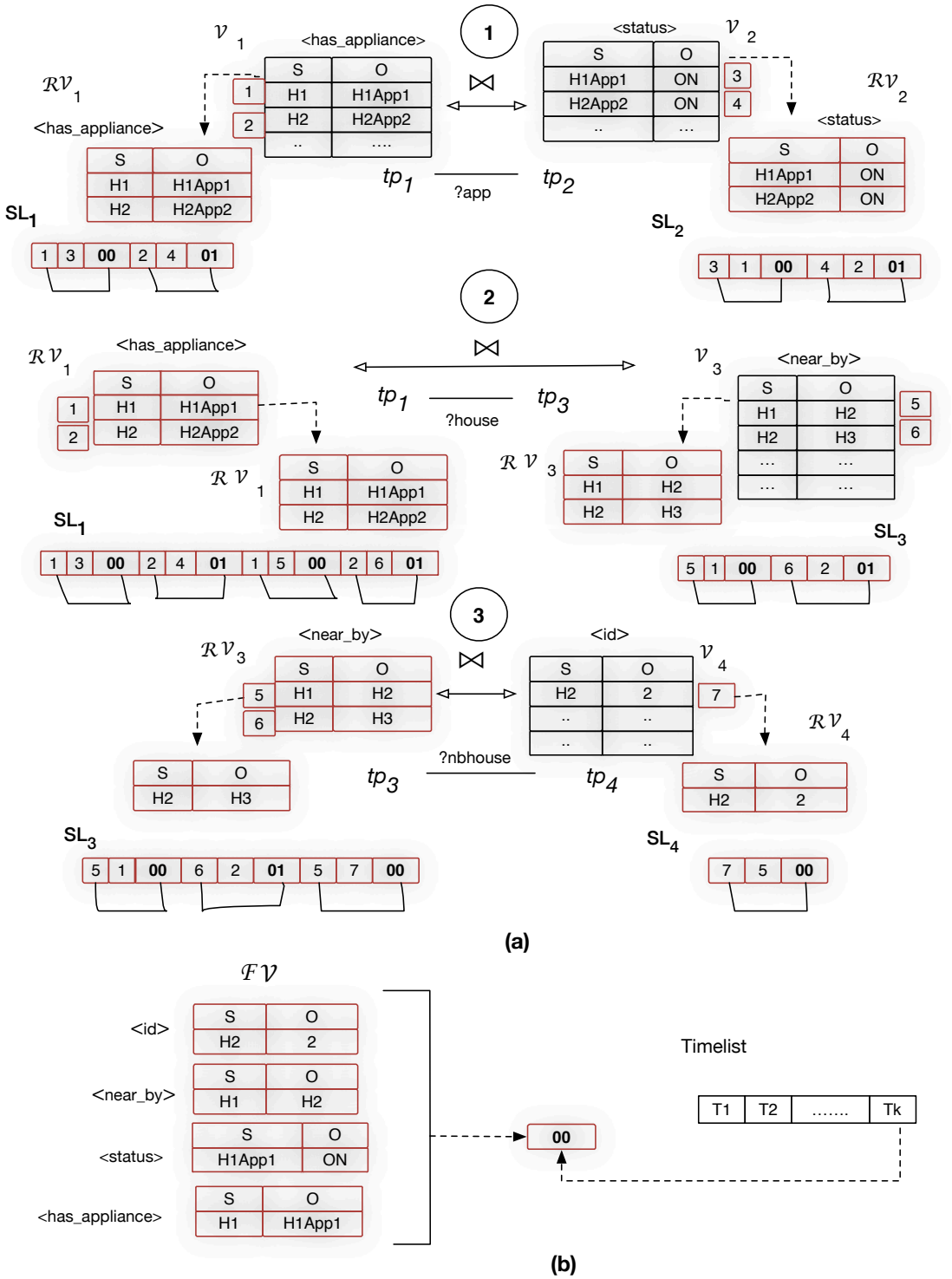

Figure 7.3: (a) Matching Process of (τ_i, G_D) with Query 6.1 as described in Example 8, (b) a set of Final Views and a Timelist

x. Note that we follow the design principle of dynamic memory allocation in our system. Hence, we work with structurally-static query conductors, i.e., views, result-views, sibling lists etc., are initialised only once (see Algorithm 3, *lines 1-5*): memory is allocated at the creation or infrequently for resizing. §Algorithm 3 first initialises the *impl-joins* set (*line 7*), which contains only the triple patterns whose views are not empty.

Algorithm 3 Query processing with QUERYPROC

```
1: τ_L ← timelist
2: viewset ← {V_tp1, V_tp2, ..., V_tp|G_Q|}
3: result-viewset ← {RV_tp1, RV_tp2, ..., RV_tp|G_Q|}
4: sibling-set ← {SL_tp1, SL_tp2, ..., SL_tp|G_Q|}
5: FV ← {FV_1, FV_2, ..., FV_|G_Q|}
6: procedure QUERYPROC(G_Q, τ_i, FV, viewset, ω, x, τ_L)
7: impl-joins ← {tp ∈ G_Q | V_tp ≠ ∅}
8: for each tp ∈impl-joins do
9:     tp_j ← getJoinTP(tp)
10:    hashJoinAndIndex(V_tp, V_tp_j, SL_tp, SL_tp_j, RV_tp, RV_tp_j)
11: end for
12: if (∀tp ∈ impl-joins, RV_tp ≠ ∅ and |impl-joins|= |G_Q| ) then
13:    FV← extractMatches(FV, sibling-set, result-viewset, τ_i, τ_L)
14:    refresh(FV, sibling-set, τ_i, ω, x, τ_L)        ▷ eviction of older events
15: else
16:    Execute INCQUERYPROC (G_Q, τ_i, FV, result-viewset, ω, xτ_L )   ▷ Section 7.5
17: end if
18: end
```

The main reason for this is that, due to the incremental nature of the algorithm there could be cases where only few views are updated and we have to incrementally join their results with previously computed ones (described later in Algorithm 4). Algorithm 3 then iterates over the *impl-joins* set. It first gets the joined triple pattern tp_j (*line 9*), which has to conduct the join operation with a certain tp. It then implements the hash-join and constructs the sibling lists for the two triple patterns by using the respective data structures. The `hashJoinAndIndex` function[1] (*line 10*) takes the viewset, result-viewset and sibling lists of the triple patterns to be joined. It either uses view ($\mathcal{V}$) or result-views ($\mathcal{RV}$) of the respective triple patterns and conducts the hash-join by iterating over the smallest one, i.e., employing the dynamic hash operator reordering, as described in Example 8. Depending upon the type of join, it then fills the intermediate result-views, while incrementally updating each sibling list with sibling tuples.

The next phase of Algorithm 3 extracts the matched results according to the sibling tuples created during the join operations. It first checks if the join operations of $tp \in G_Q$ were successful (*line 12*). If so it initiates the `extractMatches` function (*line 13*). That is, first all the distinct ordinal numbers in the smallest sibling list are extracted. Second, a depth-first search (DFS) on the sibling tuples is executed to extract all the matched subgraphs, while considering the sibling relationships between two joined triples (see Examples 7,8). Otherwise (if join operations were not successful), it sends the partially matched results to the incremental query process. The eviction of the older triples in $\mathcal{FV}$ is performed with the `refresh` operation (*line 14*); its details are provided in the next section (Algorithm 4). Furthermore, the timelist τ_L is updated with the timestamp τ_i of the

[1]The algorithm only shows the general join function, all the other types of joins (*s-s*, *s-o*, etc.,) can be implemented in a similar fashion, by either using the subject or object column of a view.

matched event and its associated ordinal number during the execution of `extractMatches`.

Complexity Analysis

Each event arrival in the window triggers three main tasks: (1) implementing multi-way joins and indexing on the set of views produced from the GraphSummary operation; (2) if all the joins produce results, matches are extracted using the sibling lists; and (3) deceased matches are removed from $\mathcal{FV}$. The cost of these operations can be described as follows. Operation (1) has two substeps, which includes linear hash-joins, and the construction of sibling lists, while propagating the ordinal numbers. Thus, if there are j join operations and l intermediate triples received for such joins, $m = l * 3$ is the size of each sibling list produced for indices. Then the total cost can be calculated as $\mathcal{O}(j(l\ log(m))) = \mathcal{O}(j(l\ log(l)))$ for j join operations: $\mathcal{O}(log\ l)$ is the average cost of binary search through a sibling list. Operation (2) utilises traditional DFS to extract the matches using the sibling-set. Thus, if g is the number of ordinal numbers, and there are p distinct pair-ids in the sibling tuples and c sibling tuples in the sibling lists, then the total cost of operation (2) can be described as $\mathcal{O}(g(p + c))$. Operation (3) consists of first collecting a set of timestamps that are outside the defined window and second deleting the (s, o) pairs associated to the timestamps. If k is the number of triples extracted during the execution of GraphSummary and stored as matches in $\mathcal{FV}$, on average, its takes $\mathcal{O}(log\ k)$ for a binary search through the timelist to extract the deceased timestamps, and there will be $\mathcal{O}(d)$ final view access and deletion operations for d deceased triples. Regarding memory complexity, QueryProc obviously requires $\mathcal{O}(k)$. The question is finally how the number of triples k grows over time. Unfortunately, a respective formal analysis would require assumptions regarding the window size, data distribution itself and how it changes and matches to a particular query over time. Even under the assumption of a static data distribution, there is no general result.

Ordering of Join Operations

The joins between the sets of triple patterns $tp \in G_Q$ are commutative and associative [PAG09b]. An efficient join ordering results in smaller intermediate results leading to a lower cost of future join operations [NW10a, Aba+09b]. Let us suppose that a multi-way join operation between views be $\mathcal{V}_1 \bowtie \mathcal{V}_2 \ldots V_{j-1} \bowtie \mathcal{V}_j$, where the cardinality measures of views, i.e., $|\mathcal{V}_i| \leq |\mathcal{V}_{i+1}|$ for every $i \in [0, j-1]$, is considered for the join operations. Then the join sequence considering the left-deep evaluation strategy [LB15] is:

$$\Big(\Big(\big((\mathcal{V}_1 \bowtie \mathcal{V}_2) \bowtie \ldots\big) \bowtie \mathcal{V}_{j-1}\Big) \bowtie \mathcal{V}_j\Big)$$

Contrary to the existing RDF processing systems, the cardinality measures of the views can change over time according to the incoming events and streams' rate. This prompts the question of dynamically re-ordering the join operations between a set of views. Recall from Section 7.3, the SummaryGraph operator captures the valid number of triples in each view before starting the execution of joins. Therefore, in Algorithm 3 and 4 (Section 7.5), we dynamically reorder the execution of join operations by considering the cardinality measures of each view after the graph summarisation process. Hence, SPECTRA is able to re-optimise the query plans on a continuos basis while a query is being processed, and it is resilient to the load variations due to the streams' rate changes.

In this section, we present three main features of the SPECTRA: (i) the SummaryGraph operator that employs the structure of the query graphs to filter data from each incoming event, (ii) the QueryProc operator that continuously joins the summarised set of views, and (iii) creation of the indices in an incremental manner during the join operations between the set of views. The matched triples within views are persisted in a set of final views to be utilised later in an incremental manner. Such a discussion is provided in the proceeding section.

7.5 Incremental Query Processing

This section provides the details of incremental query process IncQueryProc from Algorithm 1 (*line 7*). The main objectives of IncQueryProc are as follows:

1. computation of the partial matches, i.e., partially joined views are joined with the set of final views $\mathcal{FV}$;

2. the eviction of deceased matches from $\mathcal{FV}$, as it is not processed in QueryProc when IncQueryProc is executed.

The main execution of IncQueryProc is described in Algorithm 4. It first uses the timestamp of current event τ and window parameters ω and x to evict the older triples from $\mathcal{FV}$ (*line 2*), i.e., the triples whose timestamps $\tau \leq \tau_b$. The `refresh` operation consists of two steps: (i) finding the range of timestamps that are outside the window, and (ii) using the pointers of older timestamps to remove the triples from $\mathcal{FV}$. The eviction of older triples can affect the total number of matches. Therefore, instead of evaluating all the matches from scratch, we employ the sibling relationships between joined triples and their associated ordinal numbers to determine the matched graphs affected by the removal process (details are provided in Section 7.6).

Algorithm 4 IncQueryProc

```
1:  procedure IncQueryProc(G_Q, τ_i, FV, result-viewset, ω, x, τ_L)
2:  refresh(FV, τ_i, ω, x, τ_L)                          ▷ eviction of deceased events
3:  incr-tplist ← {tp ∈ impl-joins | RV_tp ≠ ∅}
4:  for each triple pattern tp ∈ incr-tplist do
5:  |  tp_J ← GetJoinTP(tp)
6:  |  if tp_J ∉incr-tplist then
7:  |  |  hashJoinAndIndex(RV_tp, FV_tp_J, SL_tp)   ▷ Use RV_tp and FV to implement the join.
8:  |  end if
9:  end for
10: if for all tp ∈incr-tplist, RV_tp ≠ ∅ then
11: |  FV ←extractMatches(FV,sibling-set,result-viewset,τ_i, τ_L)
12: end if
13: end
```

Algorithm 4 then collects (in *incr-tplist*) all the triple patterns with non-empty intermediate result-views and iterates over it to conduct the joins with the set of final views $\mathcal{FV}$. It ignores all the other joins that were previously done during the execution of QueryProc. The remaining joins are conducted using $\mathcal{RV}_{tp}$ with the corresponding views in $\mathcal{FV}$ (*line 7*). If they all produce a non-empty intermediate result-view, the (s, o)-pairs are collected (same as in Algorithm 3) and added to the $\mathcal{FV}$ set with the timestamp

of the event in τ_L (*lines 11-13*). Note that the analysis and cost of IncQueryProc operations can be directly inferred from QueryProc operations (Section 7.4.2).

Example 9 *Consider the same* Query 6.1 *and the set of final views $\mathcal{FV}$ collected in Example 8. Now consider the execution of* Query 6.1 *over G_D^j (see Figure 6.1). As described in Figure 7.4(a), two views $\mathcal{V}_1$ and $\mathcal{V}_2$ are materialised for triple patterns tp_1 and tp_2: data for other triple patterns are not present in G_D^j. Thus, the execution of* QueryProc *produces the partial matches (① in Figure 7.4(a)), which have to be matched with $\mathcal{FV}$ to preserve the property of incremental evaluation. For such a task, we determine (i) the number of joins of tp_1 and tp_2, (ii) how many joins have already been executed after the* QueryProc *operations, and (iii) how many joins need to be further executed with $\mathcal{FV}$. As shown in* Query 6.1*, tp_1 has two joins: o-s join with tp_2 and s-s join with tp_3, while tp_2 has one s-o join with tp_1. During the execution of* QueryProc*, tp_1 has already computed the join with tp_2 using views $\mathcal{V}_1$ and $\mathcal{V}_2$; the same goes for tp_2, as shown in Figure 7.4(a). Thus, we need to only implement the s-s join between $\mathcal{RV}_1$ of tp_1 and $\mathcal{FV}_2$ of tp_3. If the join produces a non-empty result-view then we add the partially matches from $\mathcal{RV}_1$ and $\mathcal{RV}_2$ in $\mathcal{FV}$. The whole process is shown in Figure 7.4(a), where $\mathcal{RV}'_1$ is the result obtained by joining $\mathcal{FV}_2$ with $\mathcal{RV}_1$ (② in Figure 7.4(a))). Finally, utilising the same technique in Algorithm 3, all the (s, o) pairs and their respective sibling relationships that are associated to the same ordinal numbers are added to the respective final views in $\mathcal{FV}$. Figure 7.4(b) presents $\mathcal{FV}$ after the* IncQueryProc*, final views $\mathcal{FV}_3$ and $\mathcal{FV}_4$ are updated with new (s,o) pairs. As the partially joined results are added in these final views, we use a sibling relationship between the ordinal numbers* `(00,110)` *for the (s, o) pairs* `(H1App2,ON)` *and* `(H1,H1App2)` *in $\mathcal{FV}_3$ and $\mathcal{FV}_4$ respectively. Such sibling relationships between ordinal numbers are efficiently utilised to incrementally update matches after the* `refresh` *operation (as described below).*

7.6 Processing Timelist and Matched Results

In this section, we presents the details of (i) how the timelist is queried to discard the events that are outside the window's boundaries, (ii) how the query matches are updated after the removal of older matches, and (ii) what are the semantics of the output matches.

Dealing with Timelist: A timelist ordered in a monotonically increasing order of timestamps is used to locate the events that are outside the defined window. We use a binary search algorithm to efficiently manage the range of values that are outside the window. Given window size ω, slide x and timestamp τ_i of the current event, we calculate τ_b and τ_e as shown in Section 3. Using τ_b, we use a binary search through the timelist, such that we are inserting τ_b in the list. If the insertion point (*index* of the timelist) is positive, it means that we have found the exact place of τ_b and all the values from $0 \leftrightarrow$ *index*-1 are older than the τ_b of the window and must be evicted. If the insertion point is a negative number (except (-1)), it means we found the place where τ_b should be inserted. Thus, all the timestamps from $0 \leftrightarrow$ (-*index*)-2 are outside the window. Finally, if the insertion point is -1, then it means that the *index* is before the start of the timelist and all the timestamps in timelist are within the defined window.

Incrementally Updating Query Matches: When a set of triples associated with an ordinal number and a timestamp is evicted from a window, all the existing matches in $\mathcal{FV}$ are updated accordingly. Suppose that a triple/triple set associated with an ordinal number g_i in $\mathcal{FV}$ is evicted from the window, then there could be two cases: (1) $\texttt{sibling}(g_i) = \varnothing$,

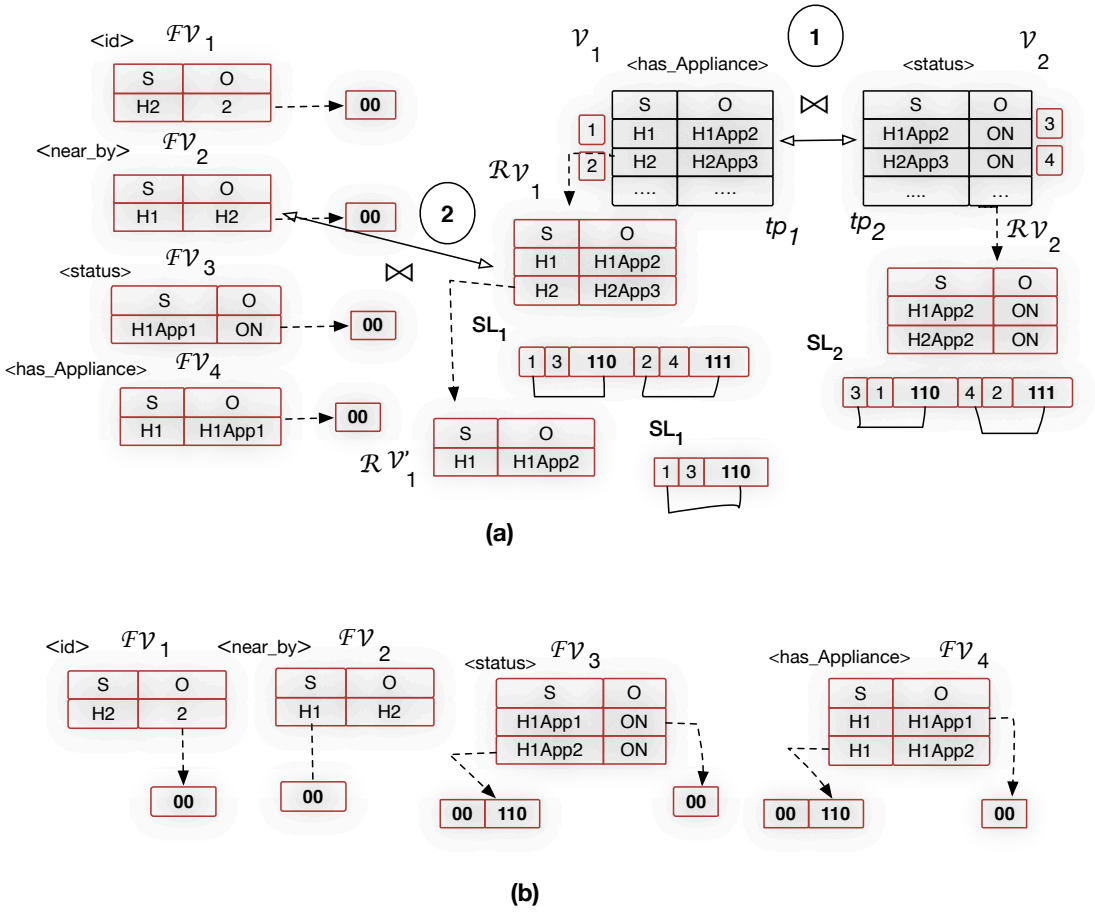

Figure 7.4: Incremental Processing of matched results of (τ_j, G_D) in Figure 6.1 with Query 6.1, as described in Example 9.

where `sibling` is a function, which determines all the sibling ordinal numbers of g_i, (2) $\texttt{sibling}(g_i) \neq \varnothing$. Case (1) describes that there are no matches effected by the removal of such ordinal number, as it does not have any siblings. Case (2) illustrates that there could be matches affected by the removal of g_i. Thus, the case (2) is incrementally handled as follows: if $\forall \texttt{sibling}(g_i), \exists (s,o) \in \mathcal{FV}_i$, $\forall i \in \{1, \ldots, |G_Q|\}$, then the existing matches are not affected by the remove operations. Otherwise, the siblings of partially matched (s,o) pairs are no longer within a window, and thus cannot be included in the output matches. Figure 7.4(b) represents a set of final views, where the removal of triples associated with the ordinal number `00` would invalidate the partially matched results in $\mathcal{FV}_3$ and $\mathcal{FV}_4$. Thus, those triples cannot be included in the matched results output.

Semantics of Output Matches: Due to the incremental nature of our matching algorithm, the semantics of matches produced by a query at time τ_i for an event (τ_i, G_D) can easily be explained through an `Istream` operator from Continuous Query Language (CQL) [ABW06]. That is, for each evaluation of QueryProc or IncQueryProc, only the newly found matches are reported in the output stream. This results in a less verbose output as compared to producing all the new and the old matches (that have already been reported) for each match execution.

7.7 Experimental Evaluation

This section presents an experimental evaluation that examines whether SPECTRA's incremental indexing and evaluation strategies are competitive as compared to the general indexing and re-evaluation based solutions.

7.7.1 Experimental Setup

Datasets and Queries: We used two synthetic benchmarks and one real-world dataset, and their associated queries for our experimental evaluations.

LUBM[2] is a widely used synthetic benchmark for benchmarking triple stores, and considers a university domain, with types like `UndergraduateStudent`, `Publication`, `GraduateCourse`, `AssistantProfessor`, to name a few. Using the LUBM generator, we create a dataset of more than 1 billion triples with 18 unique predicates. Concerning the queries, the LUBM benchmark has provided a list of queries. But many of these queries are simple 2-triple pattern queries or they are quite similar to each other. Hence we chose 7 representative queries out of this, as published in [AC10]; these queries range from simple star-shaped to complex cyclic patterns (see Appendix A.1).

The Social Network Benchmark (SNB) [Erl+15] is a synthetic dataset containing social data distributed into streams of GPS, posts, comments, photos, and static data contain the users profiles. It contains information about the persons, their friendship network and content data of messages between persons, e.g, posts, comments, likes etc. We generated a total of 50 million triples that contain data for 30,000 users. The distribution of the dataset is described in Table 7.1. For query graphs, we randomly generated three different query graphs of different characteristics by varying the types of joins and data selectivity. These query graphs are based on the use cases described in the benchmark, and describe the relationships between posts, comments, forums and persons. For example SNB-Q1 (see Appendix A.2) retrieves the post, its creator and its tag name for SNB events. These query graphs are presented in Appendix A.2. In order to compare the systems based on a triple stream model, we also reused the query graphs from LSBench[3]. It is also a social network benchmark. However, the SNB dataset presents a more connected graph structure with a larger number of attributes compared with LSBench. Thus we can introduce more complex events for LSBench queries. These query graphs are presented in Appendix A.3.

Table 7.1: Dataset Distribution for the SNB Dataset, Min and Max describe the Range of Number of Triples for each Event.

Dataset(streams)	Min (triples/event)	Max (triples/event)
500P	783	148K
1KP	2340	397K
5KP	217K	301K
10KP	50K	805K
20KP	115K	1.9M
30KP	145K	3.2M

[2]LUBM Benchmark: `http://swat.cse.lehigh.edu/projects/lubm/`, last accessed: July, 2016.

[3]LSBench Dataset and Queries: `https://code.google.com/archive/p/lsbench/`, last accessed: July, 2016.

The *SEAS*[4] project provides a real-world dataset[5] containing the power consumption statistics of family houses. It contains a set of power related attributes such as, measurement instrument types, voltages, watt values, etc. The dataset is available as RDF Data Cube and is mapped using the SEAS ontology[6]. The dataset is composed of power measurement values of a family house over the period of three years with a sampling period of 5 minutes. In total, it contains around 65 million triples. For queries, we generated one selective and one non-selective query from SEAS use cases; queries are illustrated in Appendix A.

Competitors: To compare against static triple stores, we chose two openly available and widely known systems[7]: RDFox [Nen+15] and Jena [Car+04]. Both of these systems are in-memory, and can be utilised for stream processing scenarios. For RSP systems, we selected CQELS [LP+11] and C-SPARQL [Bar+10a]. Both systems are widely used in the Semantic Web community and often compared in the literature even if both differ in their semantics: CQELS is based on push-based semantics and C-SPARQL is based on pull-based semantics. Our system resembles with CQELS due to its push-based semantics, i.e., queries are processed as soon as a new triple enters the system. In order to only compare the data structure and indexing technique of SPECTRA, we have also implemented a re-evaluation based version of SPECTRA, i.e., only executing the QueryProc operator; persisting all the data in the views; and recomputing the joins for the whole window. It is denoted as S-Rev, while the general incremental version is denoted as S-Inc in the rest of the discussion.

Settings: The performance of query processing over sliding window depends on the window size ω; we intuitively expect it to spend more time on larger windows. For all the experiments (except for S-Inc-20), we use one triple for each event, the reasons are as follows: (1) CQELS and C-SPARQL only supports single triple streams; (2) this is the worst-behaviour in terms of query performance, as a large number of matching processes are executed, one for each triple. All the experiments were performed on an Intel Xeon E3 1246v3 processor with 8MB of L3 cache. The system is equipped with 32GB of main memory and a 256Go PCI Express SSD. It runs a 64-bit Linux 3.13.0 kernel with Oracle's JDK 8u05. For robustness, we performed 10 independent runs, and we report median time and memory consumptions. SPECTRA is implemented in Java and is openly available[8].

7.7.2 Evaluation

The main objectives of our experimental evaluation are as follow:

- Detecting the effects of incremental indexing and query processing compared with the traditional static RDF and RSP solutions.
- How incremental evaluation behaves in comparison with the re-evaluation strategy?
- The effects of sliding window on query processing, and how effective is SPECTRA in evicting the deceased matches?

[4]SEAS Project: `https://itea3.org/project/seas.html`, last accessed: July, 2016.

[5]SEAS Dataset: `http://sites.ieee.org/psace-idma/data-sets/`, last accessed: July, 2016.

[6]SEAS Ontology: `http://bit.ly/1UxxLXu`, last accessed: July, 2016.

[7]Note that we do not use the commercial systems such as Node4j [**neo4j**] and Virtuoso [**virtuo**]. Neo4j is based on the property graph model and do not directly support SPARQL queries, while Virtuoso's open source version does not provide the access to the main memory system's source code; thus it is not possible to use it as a continuously query system.

[8]SPECTRA Framework: `http://spectrastreams.github.io/`, last accessed: August, 2016.

Table 7.2: Throughput Analysis ×1000 triples/second (rounded to the nearest 10) on LUMB Dataset and Queries over three different Tumbling Windows. Boldface for the Incremental Evaluation and Best Throughputs for Re-evaluation are italicised. (•) indicates Aborted Execution due to Timeouts

Window	10 sec						
Queries	Q1	Q2	Q3	Q4	Q5	Q6	Q7
C-SPARQL	12	54	8	27	76	88	19
Jena	35	80	15	68	150	110	26
CQELS	34	113	21	95	220	125	38
RDFox	28	84	23	89	123	106	32
S-Rev	*60*	*750*	*46*	*356*	*846*	*796*	*64*
S-Inc	**81**	**848**	**54**	**425**	**980**	**925**	**73**

Window	50 sec						
Queries	Q1	Q2	Q3	Q4	Q5	Q6	Q7
C-SPARQL	10	36	6	19	44	58	11
Jena	14	53	9	37	108	73	15
CQELS	22	86	13	46	143	89	21
RDFox	23	77	*17*	34	93	62	24
S-Rev	*36*	*322*	15	*88*	*637*	*748*	*28*
S-Inc	**52**	**567**	**26**	**180**	**876**	**853**	**50**

Window	100 sec						
Queries	Q1	Q2	Q3	Q4	Q5	Q6	Q7
C-SPARQL	•	•	•	12	20	22	•
Jena	•	11	•	15	33	24	•
CQELS	8	30	5	28	39	31	6
RDFox	*13*	56	*10*	31	44	38	*15*
S-Rev	11	*80*	7	*56*	*161*	*179*	10
S-Inc	**30**	**241**	**18**	**146**	**540**	**415**	**27**

- The memory overheads of our data structures and existing solutions.

Relative Performance

The first question we investigate is *How does the incremental evaluation and indexing techniques perform as compared to the re-evaluation-based techniques?* This measures the performance gain while utilising SPECTRA. For this set of experiments, we use tumbling windows: when window size $\omega = x$ is equal to the slide granularity x, the sliding window degenerates to the tumbling window. The main reasons for using tumbling windows are as follows: (1) as the window does not slide incrementally, it can provide the measures of S-Inc overheads; (2) implementing sliding windows over existing in-memory stores is a complex task, thus can offer S-Inc an unfair advantage; (3) it can effectively provide a break-even analysis, where S-Inc outperforms S-Rev. Note that, to avoid any unfair advantage, we utilise the GraphSummary operator on top of all the evaluated systems.

Table 7.2 shows the throughput analysis (higher is better) for the LUBM dataset on queries 1-7 over three window sizes. A higher throughput represents better results for S-Inc, as compared to RDFox, Jena, CQELS and C-SPARQL. The selectivity of queries has a direct impact on the number of triples added and the number of matches found in a defined window.

We start our analysis from the highly non-selective queries *LUBM-Q1*, *LUBM-Q3* and *LUBM-Q7*, each has a large number of triples associated with the triple patterns, and contains complex and cyclic patterns. Even with the GraphSummary operator, a large number of triples are inserted into the window and this results in higher cost of query computation for each matching operation. Therefore, S-Rev for large windows performed slightly worse than RDFox; for each event, the matching process results in the reconstruction of indices. RDFox with its parallel, lock free architecture and one table indexing involves with few index updates, while CQELS requires substantial updates to its B+-tree indices for each triple in the stream. S-Inc on the other hand, move the matched triples to $\mathcal{FV}$ and the new events are only matched with the respective views in $\mathcal{FV}$, without re-constructing all the indices from scratch. C-SPARQL and Jena do

not scale well with the increase in the size of the window, as their underlying storage structure (property tables) becomes quite dense for a large number of triples; C-SPARQL uses Jena and Esper [BV10] for its underlying execution model.

For selective queries *LUMB-Q4*, *LUBM-Q5* and *LUBM-Q6*, there are less triples in each window; most of the unrelated triples are pruned by our GraphSummary operator. Query *LUBM-Q5* only contains 2 triple patterns and thus has even a smaller number of triples. Therefore, S-Rev performance on these queries is comparable to S-Inc, break-even analysis of both is provided later. From the rest of the systems, CQELS is a clear winner for smaller windows with its adaptive indexing technique and its Eddy operators provide optimal query plans. However, its performance degrades with the increase in the number of triples within a window. RDFox performs better than Jena and C-SPARQL due to its parallel architecture. Query *LUBM-Q2* is less restrictive than the defined above; however, it only contains two triple patterns with one join. Thus, even with the increase in the number of triples all the systems perform better. This is due to the less number of joins required for less number of triple patterns. That is, as described in Throrem 6.1, the larger the number of triple patterns in a query graph the more number of patterns have to be matched with the incoming triples.

In the next set of experiments, we use the SNB dataset and queries to measure the performance and scalability of different systems. The main objective of these experiments is to determine how different systems scale by varying the number of triples within each event. That is, how these systems behave in batch mode, where a large set of triples are processed at once. We generated multiple distributions/streams of the SNB dataset, by changing the number of persons for one year worth of data. We divided the data into a set of events, where each event contains 1 week worth of data. Thus, each event contains a large number of triples, but there are few numbers of total events: this enabled the analysis of latency incurred due to various indexing strategies. The distribution details of the dataset are provided in Table 7.1. Events with variable sizes are processed, matches are produced and then discarded from the system. Note that, since for this set of experiments we process each event independently (using only QueryProc operator), the size of the window has not any effect on the performance. The elapsed time for each query graph was measured with warmed up cache by using the static SNB dataset (around 50K triples). We do not use CQELS for this set of experiments, as it performed poorly for larger graphs in batch mode.

We report the total execution time for SNB-Q1, SNB-Q2 and SNB-Q3 in Figure 7.5(a,b,c). SNB-Q1 contains all the triple patterns with *s-s* joins (i.e., a star-shaped pattern) and low selectivity measures, thus it returns a large number of matches. Our system scales linearly and smoothly (no large variations are observed). It outperforms others by taking the advantage of fast hash-joins and incremental indexing techniques: in a star-shaped query graph (i.e., only with *s-s* joins), the indexing process is simplified as all the triples share the same subject. Jena and Sesame, which perform better for smaller events resulted in time-outs with the increase in the size of an event; their light-weight indexing fails for large numbers of triples and their structure becomes quite dense. RDFox performs better for larger events (in batch mode) due to its parallel and lock-free architecture and one big table indexing (six columns triple table) technique. However, it resulted in high latency measure due to high creation-time for indices, as shown in Figure 7.6(b).

The query graph SNB-Q2 encompasses high selectivity measures as compared to

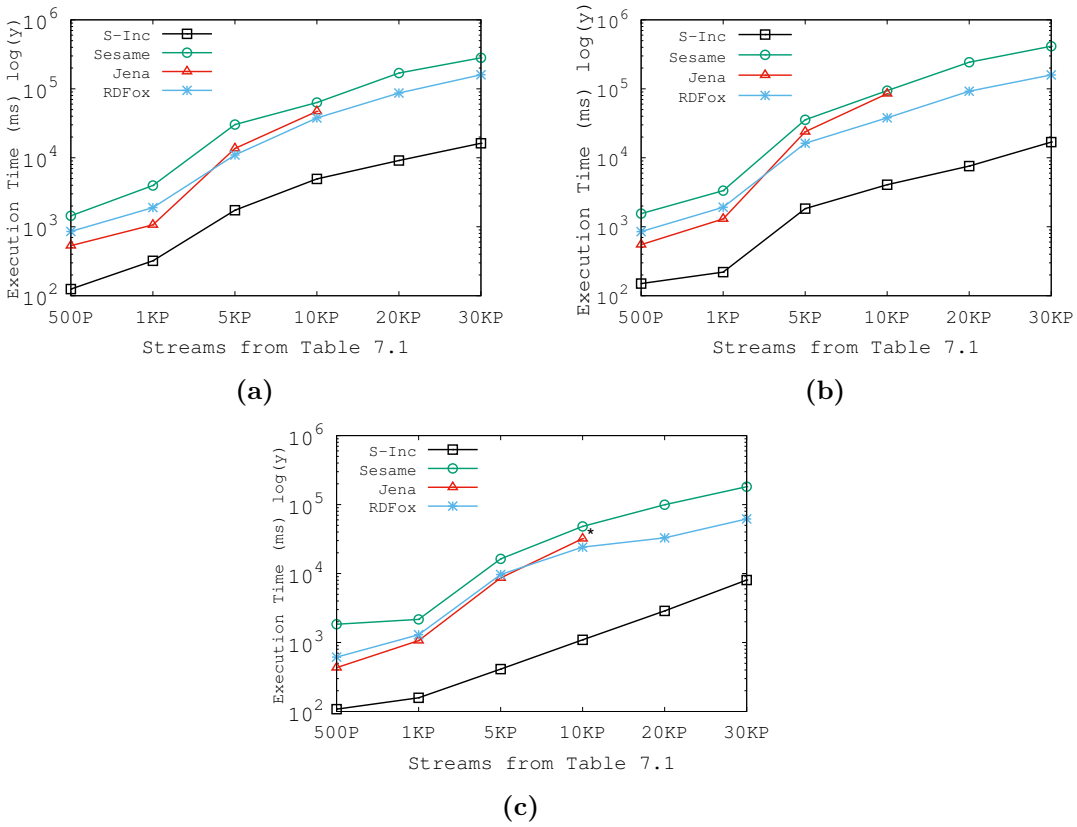

Figure 7.5: (a)(b)(c) Performance analysis of SNB Queries (1,2 and 3 respectively) (including both latency measures and query time)

SNB-Q1 and contains a combination of *s-s* and *o-s* joins, resulting into a tree-like pattern. It produces less number of matches for smaller events. However, the number of matches grows exponentially with the increase in event size. This results in an expensive exploration process over the indices produced for SPECTRA. However the lower latency values compensates for it. Jena and Sesame perform in a similar fashion for SNB-Q1 and does not scale well with the increase in the number of matches. RDFox, as compared to them, still proved to be the winner; the parallel and lock free architecture of RDFox enables parallel join operations. The query graph SNB-Q3, as compared to SNB-Q1 and SNB-Q2, contains very high selectivity measures, thus only a handful of matches are produced. Since SPECTRA only indexes the triples that are able to join, it results in higher throughput compared with other systems. Other systems first employ indexing over all the pruned triples and then execute the join process, this result in higher insertion and computation times.

In order to demonstrate the difference between our incremental indexing approach and offline indexing approaches, we report the latency measures and query time for SNB-Q1 in Figure 7.6(a,b) respectively. From earlier observations, our system takes less time to load triples from each event, as indexing is performed incrementally during query execution. However, the query time is lower for RDFox with its complex indexing technique, with high insertion/indexing time. Note that these experiments were performed over a large batch of events, thus there are less number of query evaluation calls to each systems, and each call contains a large number of triples. For such reason, RDFox query time is

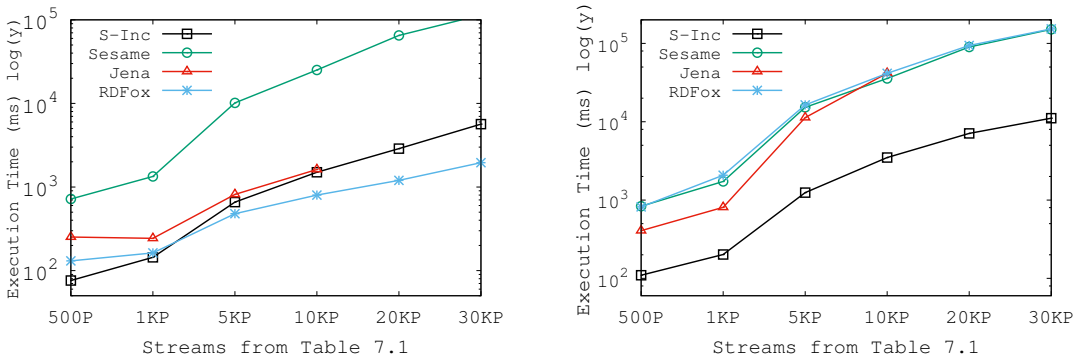

Figure 7.6: (a) Query time and (b) Latency measures of SNB-Q1 on the SNB dataset.

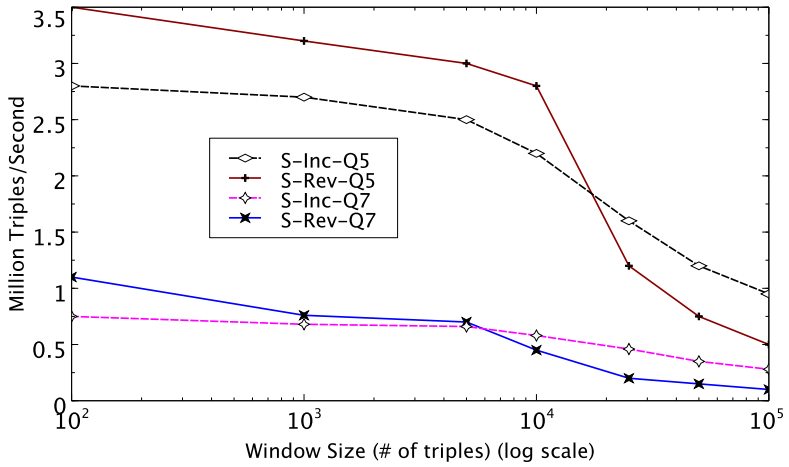

Figure 7.7: Break-Even point for the re-evaluation and incremental methods

better than SPECTRA. However, in general streaming settings with a large number of calls, as shown in earlier experiments, SPECTRA outperforms RDFox.

Break-Even Point

The second question we investigate is *What is the smallest window size at which the incremental evaluation pays off?* With a very small window, the re-evaluation strategy does so little and contains such a small number of triples that it outperforms the incremental scheme. However, with the increase in the size of the window, S-Rev becomes so expensive that it is outperformed by the S-Inc. We ran both implementations at different window sizes (for tumbling windows) and measured the throughput.

Table 7.2 shows the comparative analysis of both strategies. Due to the large size of the windows, S-Inc shows superior performance for all the queries. Figure 7.7 shows the comparative analysis on relatively smaller window from 10^2 triples to 10^5 triples; for the sake of brevity we use the number of triples for the window size. We use the selective query *LUBM-Q5* and the non-selective complex query *LUBM-Q7* from the LUBM benchmark for this analysis. S-Rev performs less operations, thus the overhead of S-Inc is, as expected to be, higher than that of re-evaluation strategy. In most of the queries, S-Inc breaks-even for relatively small window sizes (between 10^4 and 10^5

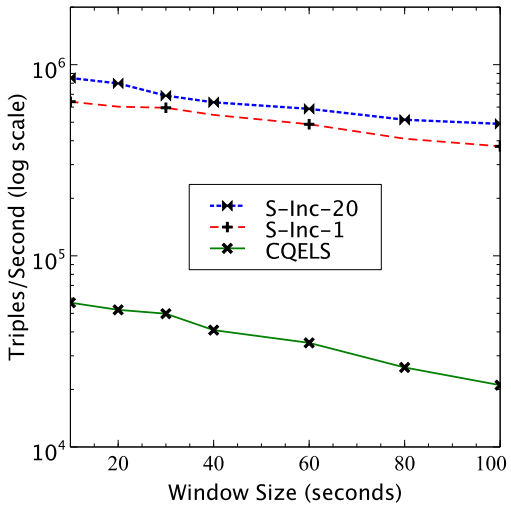

Figure 7.8: Performance of the non-selective SEAS-Q1

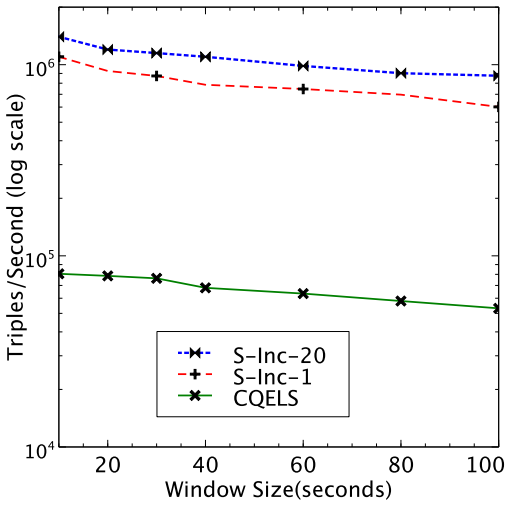

Figure 7.9: Performance of the selective SEAS-Q2

for selective queries and between 10^3 and 10^4 for non-selective queries), conforming the particle utility of S-Inc on reasonable window sizes.

Figure 7.7 shows that for $Q5$ S-Inc is about 20% slower on small window sizes and almost 3× faster on large window. Such slight slowdown is not much of the concern: at such small window sizes, extracting matches is unlikely to be the bottleneck of the application. On the other hand, S-Inc yields large speed ups even at the moderate window sizes of 10^4 triples and we can observe even larger speed-ups when further increasing the window size.

Sliding Windows

Next we investigate *How S-Inc performs when the window slides with variable granularity for triple and RDF graph streams?* This measures the performance of the system when it constantly updates its window contents. For this set of experiments, we use a slide granularity of $x = 1$, i.e., each time the matching process fires, it handles the insertion and eviction of triples. This is the worst-case behaviour for sliding windows in terms of per-event cost.

We first use the SEAS dataset (to compare different systems on real-world settings) and its non-selective (*SEAS-Q1*) and selective (*SEAS-Q2*) queries for this set of experiments. Figures 7.8 and 7.9 show the performance of SEAS queries. Note that we only use CQELS for comparative analysis, as C-SPARQL is much slower, as confirmed by our earlier experiment. Moreover, here we differentiate S-Inc-1 and S-Inc-20 by having different number of triples in each event. That is, S-Inc-1 denotes one triple per event, while S-Inc-20 denotes 20 triples per event. For both queries *SEAS-Q1* and *SEAS-Q2*, S-Inc-1 is much faster than the CQELS, which performs re-evaluation and uses adaptive indexing and operator reordering. As the window grows, S-Inc-1 is nearly an order of magnitude faster than CQELS; the number of matches and triples in a window grows linearly (specially for *SEAS-Q1*) and therefore the cost of scanning all the triples is quite high for CQELS. Furthermore, it is very expensive to scan the large number of matches with the eviction of older triples. S-Inc, with the eviction of triples from the window, does not re-evaluate the query on the remaining triples; instead ordinal numbers and

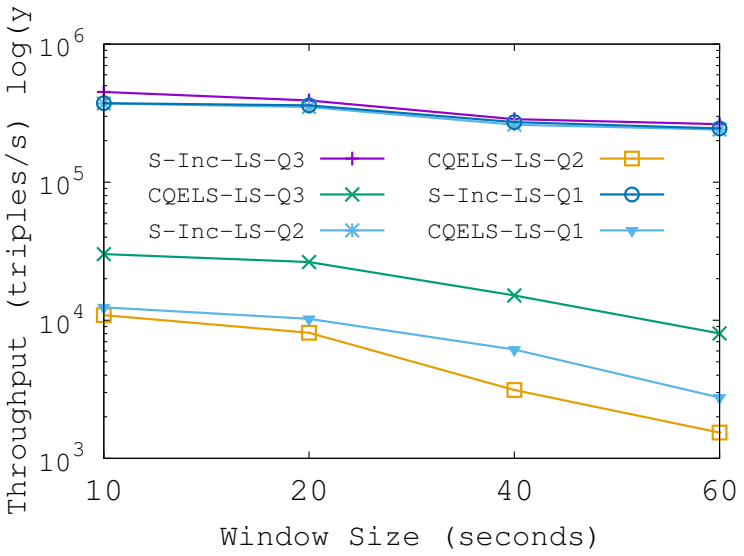

Figure 7.10: S-Inc comparison with CQELS for SNB data set and LSBench Queries

sibling relationships are utilised to determine the invalid matches. The performance of selective query *SEAS-Q2* (Figure 7.9) shows the power of SummarGraph operators, where even with the increase in the size of the window, only the triples that conform to the selective values of the query are added to the window.

Recall from Section 6.3, SPECTRA uses a general RDF graph model for the events, which allows a set triples to be enclosed in each event. The importance of this model, in terms of performance, is illustrated in Figure 7.8 and 7.9, where S-Inc-20 uses events each of 20 triples. The total number of distinct matching operations for S-Inc-20 is less than S-Inc-1, where a new matching operation is for a batch of 20 triples, instead of 1 triple. This results in 25-35% increases in performance of the system for S-Inc-20 as compared to S-Inc-1. Furthermore, this also complies to the general streaming setting, where each event consists of a set of attributes that can be related to a set of triples for the attributes set in RDF graph streams.

For the next set of experiments, we use the SNB dataset and LSBench queries to show the comparative performance measures between S-Inc and CQELS. We use triple streams (S-Inc-1) for this set of experiments, where each event contains one triple. Moreover, to showcase the importance of our SummaryGraph operators, we do not use it over CQELS for this set of experiments. The results of *LS-Q1*, *LS-Q2* and *LS-Q3* are reported in Figure 7.10. Due to the simplicity of the triple stream model, LSBench queries and the incremental evaluation strategy, our system shows similar performance measures for all queries. To elaborate, *LS-Q3* is highly selective and there are less matches and less join operations. Therefore, our system and CQELS perform much better on *LS-Q3*. However, CQELS, contrary to our system, does not prune the unwanted triples before starting the join operations; hence our system results in superior throughput. The CQELS performance degrades on less restrictive query graphs (*LS-Q1* and *LS-Q2*), where a larger number of matches are produced over a large number of triples within a defined window. Thus, for each new triple update it re-evaluates all the matches from the triples set within a window: the information about the partial and complete matches are not stored and utilised for future match operations. Furthermore, it results in time-outs

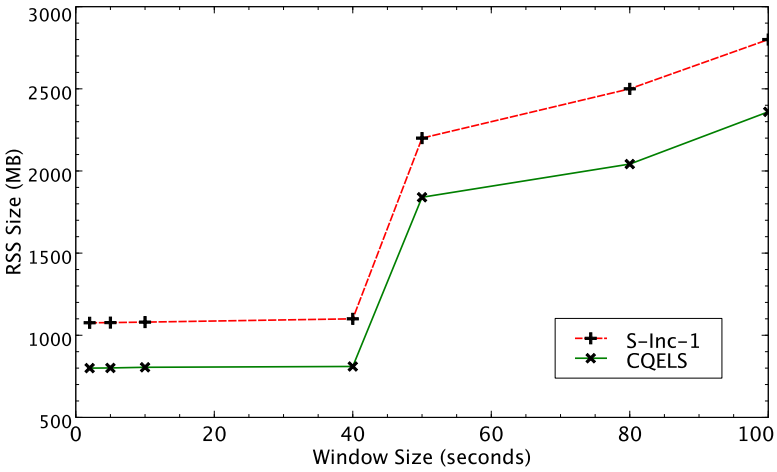

Figure 7.11: Resident set size (in MB) of S-Inc-1 and CQELS for SEAS $Q1$

for larger windows, as it continuously updates its B+-tree indices and uses a static dictionary approach. In our system, on the contrary, any change made by the newly arrived triples to the query graph matches is stored, and information about the partial and total matches are incrementally updated where needed.

Memory Consumption

What is the effect of the S-Inc data structure on the memory consumption? Figure 7.11 shows the resident set size (RSS; lower is better), which is measured using a separate process that polls the `/proc` Linux file system, once a second. Note that this method did not interfere with the overall timing results, from which we concluded that it did not perturb the experiments. Figure 7.11 shows the comparative results of CQELS and S-Inc-1, and as expected, S-Inc takes slightly more memory than CQELS due to its bidirectional multimaps. However, this pays-off with the performance improvements. At small or moderate window sizes, the impact of window size is fairly minor on RSS as compared to base RSS of the entire process. Both systems continue to consume space linearly in size of the window.

7.8 Extending SPECTRA

As discussed in Section 7.5, SPECTRA supports the output of matches in an incremental manner. But it can be extended to (i) support the `Rstream` operator [ABW06] to output all the matches that exist in a window at a certain time τ_i, (ii) to implement the out-of-order stream model.

Extension of Output Semantics: The extension of the output model for SPECTRA is a straightforward procedure. As noted in Sections 7.4 and 7.5, matched triples from each event are stored in a set of final views $\mathcal{FV}$. Thus, in order to extract all the matches ($[\![G_Q]\!]_{R_W(\tau_i)}$) at each time τ_i for a query graph G_Q, each match operation can visit all the available final views and extract incrementally stored triples. The results produced in this settings can be more verbose: the same matches can be present in the output that are computed at different evaluation times. Nevertheless, it can satisfy the semantics of the `Rstream` operator from the CQL [ABW06]: such semantics are used by the C-SPARQL

engine for processing triple streams. Note that due the verbose nature of such operators there will be increase in the load on the system's thread which is responsible for delivering the results to the defined application or the disk storage.

Out-of-order Streams: SPECTRA makes the general assumption that the events arrived within a streams are totally ordered. This assumption is not only considered by all the RDF stream processing systems, but also most of the DSMSs comply to it. The total order assumption might not be met in practice because of network latencies and distributed data sources. Therefore, in case of out-of-order streams, the system can buffer the input events for a certain maximum amount of time and then reorder them [Li+08].

7.9 Summary

In this chapter, we provided some of the answers to the following question. *How to design a scalable RDF Graph stream processing system?* Based on the limitations of existing RSP and static RDF processing systems (as discussed in the previous chapter), we proposed a new system called SPECTRA. It uses a set of vertically partitioned views to collect the summarised data from each event and employs sibling lists to incrementally index the joined triples between views. The matched results are persisted in a set of final views, thus enabling the incremental evaluation with the arrival of new events. Our contributions for this chapter as follows:

- **SPECTRA Framework.** We detailed the design of the SPECTRA framework and how it caters the limitations for existing RSP and static RDF graph processing systems.
- **Query-based Graph Summarisation.** We provided a light-weight query-based graph summarisation technique to prune the unwanted triples from each RDF graph event.
- **Incremental Indexing.** We proposed an incremental indexing technique, where triples within views are indexed during the join process.
- **Incremental Query Processing.** We proposed two query processing operators, where the QueryProc matches a set of triples within an event with the defined query graph, and the IncrQueryProc uses already computed matches to process newly arrived events using openly available datasets and systems.
- **Experimental Evaluation.** Given these properties, the experimental results show that our proposed techniques clearly outperform traditional offline/online indexing and re-evaluation based solutions.

An RDF graph stream processing system that employs customised optimisation while considering properties of both RDF graph and streaming environment can provide considerable advantage over existing approaches. In this chapter, we demonstrated such observation. In the next chapter, we move from the stateless query processing to a stateful one, where temporal operators come into action to achieve temporal pattern matching.

Part III

Semantic Complex Event Processing: Model, Language and Implementation

Any change in the true wind will show its fingerprint in the sea. A fresh train of ripples or waves will run a web over waves caused by the true wind.

— Francis O'Neill

8

A Query Language for SCEP: Syntax and Semantics

In this chapter, we transit from the topic of semantically-enabled stream processing to Semantic Complex Event Processing (SCEP), while providing the syntax and semantics of our SCEP query language called SPAseq*. We first describe the limitations of the existing SCEP languages, and motivate the requirement of a new one. Second, through intuitive use cases, we present the main constructs of* SPAseq*. Third, we provide a qualitative analysis of* SPAseq *and its competitor: EP-SPARQL.*

Contents

This chapter is structured as follows: Section 8.1 presents the introductory discussion about SCEP. Section 8.2 presents the motivation of a new SCEP

language and the limitations of existing ones. Section 8.3 introduces the SPASEQ *language and its data model. Section 8.4 describes the syntax of* SPASEQ *with intuitive examples. Section 8.5 presents various complex examples and use cases that* SPASEQ *can handle. Section 8.6 provides the semantics of* SPASEQ*. Section 8.7 provides the qualitative and comparative analysis of* SPASEQ *and EP-SPARQL. Section 8.8 concludes the chapter.*

8.1 Introduction

Complex Event Processing (CEP) denotes algorithmic methods for making sense of events by deriving higher-level knowledge, or extracting complex events from lower-level events in a timely fashion. As previously discussed, CEP applications commonly involve three requirements:

1. complex predicates (filtering, correlation),
2. temporal, order and sequential patterns,
3. transforming the event(s) into more complex/composite structures.

CEP systems have demonstrated utility in a variety of applications including financial trading, security monitoring, social and sensor network analysis. Following the trend of using RDF as a unified data model for integrating diverse data sources across heterogeneous domains, Semantic CEP (SCEP) employs the RDF data model to handle and analyse complex relations over a high volume of RDF graph streams. Thus, designing an efficient query language is a vital part of SCEP: it allows users to specify known queries or patterns of events in an intuitive way, while hiding the implementation details.

There are various commonalities among SCEP languages and traditional Semantic Web data languages (such as SPARQL). There are, however, also many important discrepancies between the capabilities and premises of traditional query languages. Herein, we summarise these requirements for completeness as follows:

1. RDF events are received over time in a stream-like manner, whereas in a triple store all facts are available at once and usually stored in a persistent manner. Thus, a SCEP language requires an operator to select a specific stream to evaluate events.
2. Event streams are unbounded into future, and potentially infinite, whereas triple stores employs a finite model. In order to execute SCEP queries, events are bound by certain windows (as described in Chapter 5 (Section 5.2)). Therefore, a SCEP language is required to provide a window operator.
3. Relationships between events, such as temporal order or causality, play an important role for SCEP. In general triple stores, relationships between facts are part of the data (e.g., references through predicates and foreign keys). This is, however, not the case for SCEP, and a SCEP language requires explicit operators to capture these relationships.
4. Timing of answers has to be considered when querying events: SCEP queries are evaluated continuously against the event stream and generate answers at different times. Therefore, the evaluation semantics of output results for a SCEP query language should either be push or pull-based.

These added attributes of SCEP languages are required not only to carefully handle the expressibility of the language that can support most of the SCEP use cases, but also to balance the executional semantics with an efficient implementation.

As previously discussed, while there does not exist a standard language for expressing continuous queries over RDF graph streams, a few options have been proposed. In particular, the first strand of research focuses on extending the scope of SPARQL to enable stateless continuous evaluation of RDF triple streams. These approaches, including CQELS [LP+11], C-SPARQL [Bar+10a], SPARQL$_{\text{Stream}}$ [CCG10], are classified under RSP systems, and do not provide any operator to extract temporal relationships between events. The second strand of research focuses on extending SPARQL with stateful operators. In particular, EP-SPARQL [Ani+11] extends SPARQL with sequence constructs to allow temporal ordering over triple streams. Although, EP-SPARQL can be classified under the umbrella of SCEP, its definitions of sequence operators and graph pattern matching operators are mixed; thus, it makes it difficult to extend it for RDF graph streams (as described in Section 8.2.2). Moreover, it works on a single stream model and lacks explicit *kleene-+*, negation[1] and event selection strategies. These shortcomings led to the implementation of a new query language, called SPAseq, which is described in this chapter.

8.2 Why A New Language?

To justify the need of a new query language for SCEP, we use a running use case: it illustrates the main limitations of existing approaches and shows the kind of expressiveness and flexibility needed.

8.2.1 A Motivating Example

Consider a *smart grid* application that processes information coming from a set of heterogeneous sensors. Based on the events from these streams, it notifies the users or an online service to take a decision to improve the power usage. Let us consider, it is working on three streams: the first stream (S_1) provides the events about the power-related sources from a house, the second stream (S_2) provides the weather-related events for house, and the third stream (S_3) provides the power storage-related events. Herein, we present a simple use case (**UC**) to illustrate the features a SCEP language should provide.

UC 1 (**Smart Grid Environment Monitoring**): *Consider the above-mentioned three RDF event streams S_1, S_2 and S_3, which are fed to an application that notifies the user to switch to the stored power instead of main power supply, if the system observes the following sequence of events: (**A**) the price of electricity generated by a power source (fuel) is greater than a certain threshold, (**B**) weather conditions are favourable for renewable energy production (one or more events), and (**C**) the price of renewable energy source (solar) is less than the previous power source.*

UC 1 requires that a SCEP language should meet the following main principles.

- Since the RDF graph model is the corner-stone of SCEP, its features such as seamless integration of multiple heterogeneous streams should be considered for the design of a SCEP language.

[1] the negation operator is not an explicit part of the EP-SPARQL formalism, but can be defined with a combination of other operators.

- The main aim of a SCEP language is to provide temporal operators on top of standard SPARQL operators. Thus, the list of temporal operators (as discussed in Chapter 5 (Section 5.2)), such as sequencing, conjunction, disjunction, negation, kleene-+ and event selection strategies should be supported in a SCEP language.
- The SCEP language (as discussed in Chapter 4 and 5) should provide operators to directly enrich events through a static background knowledge.

Following the language considerations as discussed above, we also provide some general requirements for the SCEP language.

- The SCEP language should follow the principal of *genericity*, i.e., its design should be independent of the underlying execution model.
- The SCEP language should provide *simple syntax and semantics* that can easily be extended.
- The SCEP language should provide the property of *compositionality*. That is, the output of a query can be used as an input for another.
- The SCEP language should be *user-friendly* with low barrier of entrance, especially in the Semantic Web community.

The aforementioned attributes are the basic requirements for a SCEP language. Herein, using them as a yardstick we outline the limitations of existing languages, in particularly EP-SPARQL.

8.2.2 Limitations of Existing SCEP Languages

As mentioned in Chapter 5, EP-SPARQL is the only SCEP language that, to some extent, can express desirable attributes for SCEP. However, if one yanks the rug out from beneath EP-SPARQL, one can find that many important features of SCEP are not present in EP-SPARQL. These limitations of EP-SPARQL are listed as follows:

- *Multiple Heterogeneous Streams*: As previously discussed in Chapter 5 (Section 5.4), the data model of EP-SPARQL is based on a single stream model. That is, a single RDF event stream is used to evaluate the temporal sequences between events. This contradicts some of the motivations behind SCEP: the support of heterogeneous multiple streams forms the backbone of SCEP. The reason is based on its inspiration from a CEP system (ETALIS), where an RDF event stream is mapped onto Prolog object stream. Hence, its design is directly motivated from its underlying executional model, and extending it for the multiple stream requires complete overhauling of its semantics.
- *Temporal Operators*: EP-SPARQL only supports a small subset of temporal operators, and operators such as kleene-+, event selection strategies are not supported. These operators are important for many applications where semantic noise is observed (more details are provided in Section 8.5). Moreover, the conjunction and disjunction operators in EP-SPARQL are inspired from SPARQL (`OPTIONAL` and `AND`), and do not provide the nesting over a set of events as described for CEP systems. This leads to a design where the semantics of temporal operators and SPARQL graph patterns are mixed, and hence cannot be easily extended.

- *Enriching Events with Background Knowledge*: The static background knowledge is used to extract further implicit information from events. As a query language, EP-SPARQL does not provide any explicit operators to join graph patterns defined on an external knowledge and incoming RDF events. It, however, employs Prolog rules or RDFS rules within an ETALIS engine. Nevertheless, such feature should be provided at the query level to give users control on which information is required or not: this observation is based on the RSP languages that provide such functionality.

- *Compositionality and Negation Operator* The compositionality in EP-SPARQL is supported through recursion and blank nodes. Consider Query 8.1, which uses the generation of new IRIs via blank nodes in the head of `CONSTRUCT` clause. This enables an infinite number of possible triples and as the query simultaneously uses recursion: `_:aaa :hasSum ?sum` is constructed out of `?point :hasSum ?prevsum`, and according to the EP-SPARQL formalism (as described in Chapter 5) this query is most likely undecidable.

  As no explicit negation operator is provided for EP-SPARQL, it uses the complicated `EQUALSOPTIONAL` clause in conjunction with the `!Bound(?inbetween)` filter to support negations. We think this is an inelegant way of providing the negation: if there are n events match to `:ACM :hasStockPrice ?price` triple patterns (in Query 8.1) within the defined window, then the query has to keep $\sum_{i=1}^{n-1} 3i = 1.5 \times n \times (n-1)$ triples as partial matches within memory to correctly evaluate these expressions.

```
1  CONSTRUCT _:aaa :hasCount ?count .
2            _:aaa :hasSum ?sum .
3    { SELECT ?count AS ?prevcount + 1
4             ?sum AS ?prevsum + ?price
5      WHERE {{ ?point :hasCount ?prevcount .
6               ?point :hasSum   ?prevsum .
7             } SEQ { :ACME :hasStockPrice ?price . }
8             } EQUALSOPTIONAL
9             {{ ?point :hasCount ?prevcount .
10              ?point :hasSum ?prevsum .
11            } SEQ { :ACME :hasStockPrice ?inbetween .
12            } SEQ { :ACME :hasStockPrice ?price . }
13            }
14     FILTER ( !BOUND(?inbetween) &&
15              getDURATION() < "P10D"^^xsd:duration )}
```

Query 8.1: EP-SPARQL query for compositionality and Negation

8.3 The SPAseq Query Language

Considering the shortcomings of EP-SPARQL, as a part of the contribution of this thesis, we propose a new language called SPAseq. The design of SPAseq is based on the following main principles: (1) support of an RDF graph event model, (2) adequate expressive power, i.e., not only based on core SPARQL constructs but also including general purpose temporal operators, (3) genericity, i.e. independent of the underlying evaluation techniques, (4) simple syntax and semantics that can be extended (5) compositionality, i.e, the output of a query can be used as an input for another one, (6) user-friendly with a low barrier of entrance, especially in the Semantic Web community.

The most important feature of SPASEQ is that it clearly separates the query components for describing temporal patterns over RDF graph events, from specifying the graph pattern matching over each RDF graph event. This enables SPASEQ to employ expressive temporal operators, such as kleene-+, negation, optional over events from heterogeneous streams. In the following, we start with the data model of SPASEQ and then provide the details regarding its syntax and semantics.

8.3.1 Data Model

In this section, we introduce the structural data model of SPASEQ that captures the concept of RDF graph-based events, which serves as the basis of our query language. We use the RDF data model (as introduced in Chapter 2) to model an event. That is, we assume three pairwise disjoint, infinite sets $\mathcal{I}$ (IRIs), $\mathcal{B}$ (blank nodes), and $\mathcal{L}$ (literals). An RDF triple is a tuple $\langle s, p, o\rangle \in (\mathcal{I} \cup \mathcal{B}) \times \mathcal{I} \times (\mathcal{I} \cup \mathcal{B} \cup \mathcal{L})$. An RDF graph is a set of RDF triples. Based on this we reuse Definitions 6.1 and 6.2 to describe RDF graph events and streams.

Definition 8.1: RDF Graph Event

An **RDF graph event** (G_e) *is a pair* (τ, G) *where* G *is an RDF graph, and* τ *is an associated timestamp that belongs to a one-dimensional, totally ordered metric space.*

We do not make explicit what timestamps are because one may rely on, e.g., UNIX epoch, which is a discrete representation of time, while others could use `xsd:dateTime` which is arbitrarily precise.

In our setting, *streams* are simply sets of RDF graph events defined as follows:

Definition 8.2: RDF Graph Event Stream

An **RDF graph event stream** $\mathcal{S}_g$ *is a possibly infinite set of RDF graph-based events such that, for any given timestamps* τ *and* τ'*, there is a finite amount of events occurring between them.*

An RDF graph event stream can be seen as a sequence of chronologically ordered RDF graphs marked with timestamps: several RDF graph-based events can "happen" at the same time. In addition, we follow the uniqueness property for RDF graphs annotated with the same timestamps, That is, an event $(\tau, G) \in \mathcal{S}_g$ s.t $\forall \tau, \exists\,!\, G$, $(\tau, G) \in \mathcal{S}_g$. To handle multiple streams, we identify each using an IRI, and group them in a data model we call RDF *streamset*.

Definition 8.3: Named Stream

A **named stream** *is a pair* $(u, \mathcal{S}_g)$ *where* u *is an IRI, called the* stream name*, and* $\mathcal{S}_g$ *is an RDF graph event stream. An* **RDF graph streamset** Σ *is a set of named streams such that stream names appear only once.*

In the rest of the chapter, we simply use the terms *graph* for RDF graph, *event* for RDF graph event, *stream* for RDF graph event stream, and *streamset* for RDF graph

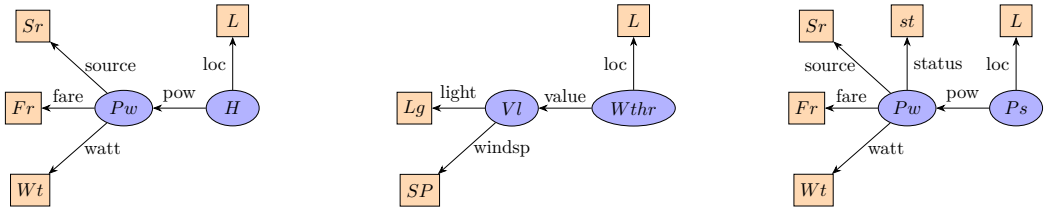

Figure 8.1: Structure of the Events from three Named Streams, (8.1a) $(u_1, \mathcal{S}_{g_1})$ Power Stream's Event, (8.1b) $(u_2, \mathcal{S}_{g_2})$ Weather Stream's Event, (8.1c) $(u_3, \mathcal{S}_{g_3})$ Power Storage Stream's Event

streamset. Moreover, we simply use $\mathcal{S}$ to denote a stream.

Example 10 *Recall* ***UC*** *1, here we extend it with our data model. The first named stream* $(u_1, \mathcal{S}_{g_1})$ *provides the events about the power-related sources from a house, the second named stream* $(u_2, \mathcal{S}_{g_2})$ *provides the weather-related events for house, and the third named stream* $(u_3, \mathcal{S}_{g_3})$ *provides the power storage-related events. Figure 8.1 illustrates the structure of the events from each source. For instance, an event from a named stream* $(u_1, \mathcal{S}_{g_1})$ *can contain the following set of RDF triples and a timestamp:*

$$\begin{aligned}(\tau_i, G_i) \quad = (\quad & 10, \{(H1, loc, L1), (H1, pow, Pw1), \\ & (Pw1, source, solar), (Pw1, fare, 5), (Pw1, watt, 20)\})\end{aligned}$$

```
1   SELECT ?house ?fr1 ?fr2
2   WITHIN 30 MINUTES
3   FROM STREAM S1 <http://smartgrid.org/house>
4   FROM STREAM S2 <http://smartgrid.org/weather>
5   FROM STREAM S3 <http://smartgrid.org/storage>
6
7   WHERE {
8
9      SEQ (A; B+, C)
10     DEFINE GPM A ON S1 {
11         ?house  :loc    ?l.
12         ?house  :pow    :Pw.
13         :Pw     :source ?s1.
14         :Pw     :fare   ?fr1.
15         FILTER(?s1 = 'fuel' &&
16                ?fr1 > 20).
17     }
18      DEFINE GPM B ON S2 {
19         ?wther  :loc    ?l.
20         ?wther  :value  :Vl.
21         :Vl     :light  ?lt.
22         :Vl     :windsp ?sp.
23         FILTER (?sp > 3 &&
24                 ?lt > 40).
25      }
26
27     DEFINE GPM C ON S3 {
28         ?storage  :loc    ?l.
29         ?storage  :pow    :Pw.
30         :Pw       :source ?s2.
31         :Pw       :fare   ?fr2.
32         FILTER (?s2 = 'solar' &&
33                 ?fr2 < ?fr1).
34     }
35  }
```

Query 8.2: A Sample SPAseq Query for the **UC 1**

8.4 Syntax of SPAseq

This section defines the abstract syntax of SPAseq, where SPAseq queries are meant to be evaluated over a streamset, and each query is built from the two main components: *graph pattern matching expression* (GPM) for specifying the SPARQL graph patterns over events; and *sequence expression* for selecting the sequence of a set of GPM expressions. For this discussion, we assume that the reader is familiar with the definition and the algebraic formalisation introduced in Chapter 2 (Section 2.4) and Chapter 6 (Section 6.3).

In particular, we rely on the notion of SPARQL graph patterns by considering operators `AND`, `OPT`, `UNION`, `FILTER`, and `GRAPH`.

Definition 8.4: SPAseq Query

A SPAseq *query is a tuple* $Q = (\mathcal{V}, \omega, \text{SeqExp})$*, where* $\mathcal{V}$ *is a set of variables,* ω *is a duration, and* SeqExp *is a* sequence expression *defined according to the following grammar:*

$$\begin{aligned} \text{SeqExp} &::= \mathit{Atom} \mid \text{SeqExp } \text{`;'}\ \mathit{Atom} \mid \text{SeqExp } \text{`,'}\ \mathit{Atom} \\ \mathit{Atom} &::= \mathit{GPM} \mid \mathit{GPM}\ [\text{`?'} \mid \text{`!'} \mid \text{`+'}] \mid (\mathit{GPM}\ \text{`<>'}\ \mathit{GPM}) \mid (\mathit{GPM}\ \text{`|'}\ \mathit{GPM}) \\ \mathit{GPM} &::= (u, P) \end{aligned}$$

where $u \in \mathcal{I}$ *is an IRI,* P *is a SPARQL graph pattern, and* (u, P) *is called a* graph pattern matching expression *(GPM).*

The concrete syntax of SPAseq is illustrated in Query 8.2 with syntactic sugars that are closer to the SPARQL. It contains three GPM expressions each identified with a variable (`A`, `B`, `C`), which allows one to concisely refer to GPMs and to the named streams. Moreover, these variables are employed by the sequence expression to apply various CEP operators and event selection strategies (';', ',').

It is not difficult to see the main property of the SPAseq language with the separation of sequence and GPM expressions. Herein, we first study how the sequence expression interacts with the graph pattern to enable temporal ordering between matched events.

The *sequence temporal patterns* between events detect the occurrence of an event *followed-by* another: it can represent various different circumstances using binary temporal operators between events. The sequence expression SeqExp in SPAseq is used to determine the sequence between the events matched to the graph pattern P. The symbols {'?', '+', '!'} are unary operators, where the optional operator ('?') corresponds to zero or one occurrence of an event; the *kleene-+* operator ('+') corresponds to the occurrence of one or more events of the same kind; and the negation operator ('!') is used to describe the non-occurrence of certain events. The symbols {';', ','} are binary operators which describe the interpretations of the sequence between events, i.e., event selection strategies. An event G_e^i matched to P_i followed-by an event G_e^j matched to P_j can be interpreted as (1) the occurrence of an event G_e^i is followed-by an event G_e^j and there being no events of any other type between them (*immediately followed-by* (',')); (2) the occurrence of an event G_e^i is followed-by an event G_e^j and there can be other events of different types (the type of an event is distinguished by the stream id u) between both events (*followed-by* (';')). That is, all the irrelevant events are skipped until the next relevant event is read for the *followed-by* operator. Finally, the conjunction and disjunction between the events is also introduced within the sequence expression through operators ('<>') and ('|') respectively. They provide the intuitive way of determining if a set of events has happened at the same time (conjunction) or only one event among the set of events has happened (disjunction).

Example 11 *Consider the* SPAseq *Query 8.2, which illustrates the* ***UC 1****. The sequence expression* `SEQ(A; B+, C)` *depicts that the query will return a match: if the events of type* `A` *and defined on a stream* `S1` *match to the GPM expression (*`GPM A`*) followed-by one or more events (using operators (';') and ('+')) from stream* `S2` *that match to the GPM expression (*`GPM B`*), and finally* immediately followed-by *(using operator (',')) an*

event from stream `S3` that matches to the GPM expression (`GPM C`). Notice that a GPM expression mainly utilises the SPARQL graph pattern P for the evaluation of each event.

8.5 SPAseq By Examples

In this section, we provide a list of complex use cases supported by SPAseq, while highlighting its SPARQL-based and CEP-based operators.

UC 2 (V-shaped Pattern) *A query with V-shaped pattern describes the sequence of values that go down to a local minimum, then rising up to a local maximum, which was higher than the starting value.*

The application of V-shaped pattern ranges from stock analysis, weather prediction to trajectory classification, where generally a kleene-+ operator is used to select the occurrence of one or more events with the similar behaviour. For instance, in **UC** 1 a user can get a V-shaped pattern for the price of generated electricity.

Query 8.3 presents a V-Shaped SPAseq query over stream of stock events. The `SELECT` expression provides the projection of various variables within the GPM expressions, while the GPM expression utilise the `?company` variable to select the company mappings, and its corresponding volume and price mappings. The V-shaped pattern can also be spiced up with the conjunction (or disjunction) operator to evaluate the occurrence of two or more events at the same time. For instance, the sequence expression in Query 8.3 can be replaced with `SEQ (A, (B<>C), C)`.

```
1   PREFIX  pred: <http://example/>
2   SELECT  ?company ?p1 ?p2 ?p3 ?vol1 ?vol2 ?vol3
3   WITHIN 30 MINUTES
4   FROM STREAM S1 <http://stockmarket/stocks>
5
6   WHERE {
7
8      SEQ (A, B+, C)
9      DEFINE GPM A ON S1 {
10      ?company pred:price ?p1.
11      ?company pred:volume ?vol1.
12     }
13      DEFINE GPM B ON S1 {
14        ?company pred:price ?p2.
15        ?company pred:volume ?vol2.
16        Filter (?p2 < ?p1 )
17      }
18
19     DEFINE GPM C ON S3 {
20      ?company pred:price ?p3.
21      ?company pred:volume ?vol3.
22      Filter (?p3 > ?p2 && ?p3 > ?p1).
23     }
24  }
```

Query 8.3: V-shaped Pattern: SPAseq query

UC 3 (Trajectory Classification) *Trajectory classification involves in determining the sequence of objects movement (trajectories) to determine their types. For instance, finding the fishing boats by discovering the spatial relations between boats over some time interval.*

A SPAseq query to determine the trajectory of fishing boats is described in Query 8.4. It represents the following sequence: **A**: vessel leaves the harbour, **B**: vessel travels by keeping the steady speed and direction (one or more events are registered with kleene-+ operator), **C**: vessel arrives at the fishing area and stops. The GPM expressions in the query employ the same `?vessel` variable to extract the defined sequences related to specific boats. Another important operator described in the query is the use of the `GRAPH` operator (from SPARQL) to join the event data with the static knowledge-base (KB). That is, using an external KB, the query extracts the name of the vessels (`?n`) and its company name

(?cname) that follow the sequence defined in the sequence expression. This enables a user to define custom patterns to enrich events with the same semantics defined for SPARQL.

```
PREFIX  pred: <http://example/>
SELECT  ?vessel ?n ?cname
WITHIN 30 MINUTES
FROM STREAM S1 <http://harbour.org/boats>

WHERE {

   SEQ (A, B+, C)
   DEFINE GPM A ON S1 {
    ?vessel pred:speed ?s1.
    ?vessel pred:location ?loc1.
    ?vessel pred:direction ?dir1.
     Filter (?loc1 = 'harbour' && ?s1 > 0 )
   }
     DEFINE GPM B ON S1 {
       ?vessel pred:speed ?s2.
       ?vessel pred:location ?loc2.
       ?vessel pred:direction ?dir2.
  Filter (?dir1 = ?dir2 && ?s2 > ?s1)
     }

   DEFINE GPM C ON S3 {
     ?vessel pred:speed ?s3.
     ?vessel pred:location ?loc3.
     ?vessel pred:direction ?dir3.


     GRAPH <http://harbour.org/db> {
     ?vessel :name ?n.
     ?vessel :operatedBy ?company.
     ?company :name   ?cname.
     }
Filter (?loc3 = 'fishingarea' && ?s3 = 0).
   }
 }
```

Query 8.4: Trajectory Classification: SPAseq query

UC 4 (Inventory Management) *It is an interesting use case for CEP/SCEP, where RFID generated events are used to track the status of a product/equipment. Consider a system monitoring the status (surgical usage, recycling, etc.) of equipments in a hospital by using various RFID sensors. Then we can define a critical event such that if a surgical tool is washed/recycled and is put back into the use without being first disinfected, then alert the required personnel.*

In order to determine the sequence described in **UC 4**, we need to track the non-occurrence of specific events though a negation operator, i.e., non-disinfection of the surgical tool. The SPAseq Query 8.5 presents such use case, and it consists of three GPM expressions. The first GPM expression (`GPM A ON S1`) determines the recycling status of an instrument, the second GPM expression (`GPM B ON S1`) utilises the same variable for the instrument (`?inst`) to determine if it has been disinfected or not, and the third GPM expression determines the status of the instrument if it has been used or not. The sequence expression (`SEQ(A, B!, C)`) orchestrates the matching of the GPM expressions, i.e., if an instrument is used without being first disinfected. The negation operator ('**!**') in the sequence expression makes sure that the sequence is only matched if there are no events between `A` and `B` such that the status of the instrument is "disinfected".

Another important property of SPAseq, which is described in Query 8.5, is its support for compositionality. That is, new RDF graph events can be constructed so that they match the defined sequence. SPAseq employs the standard `CONSTRUCTs` expression from SPARQL to create new graph-based events from the matched mappings. The set of constructed events takes the form of a stream, and they can either be transmitted to the defined sink (an application) or can be reused within the query.

```
1   PREFIX  pred: <http://example/>
2
3   CONSTRUCT S2 <http://hospital.org/newStream> {
4   ?inst pred:InRoom ?r3.
5   ?inst pred:status "non-disinfection"@en.
6   ?inst pred:name ?n1.
7   }
8  WITHIN 60 MINUTES
9  FROM STREAM S1 <http://hospital.org/instruments>
10
11  WHERE {
12
13     SEQ (A, B!, C)
14     DEFINE GPM A ON S1 {
15      ?inst pred:name ?n1.
16      ?inst pred:status ?st1.
17       Filter (?st1 = 'recycled')
18     }
19      DEFINE GPM B ON S1 {
20           ?inst pred:status ?st2.
21       Filter (?st2 = 'disinfected')
22      }
23
24     DEFINE GPM C ON S3 {
25      ?inst pred:InRoom ?r3.
26      ?inst pred:name ?n1.
27      ?inst pred:status ?st3.
28       Filter (?st1 = 'can use')
29     }
30  }
```

Query 8.5: Inventory Management: SPAseq query

8.6 Formal Semantics of SPAseq

To formally define the semantics of SPAseq queries, we use the concept of set of mappings as defined in [PAG09b]. We use the standard join ($\bowtie$), union ($\cup$), minus ($\setminus$), optional ($\leftouterjoin$) and projection (π) operators over a set of mappings, and we also make use of the semantics of SPARQL graph pattern P over an RDF dataset as defined in Chapter 2. In particular, we use the definition of the evaluation of graph patterns over an RDF graph G as a function $[\![.]\!]_G$ that, given a graph pattern P, returns a set of mappings denoted as $[\![P]\!]_G$ (Definition 6.6 in Chapter 6).

8.6.1 Rough Work

Before describing the semantics of SPAseq, herein, we first present an intuitive way of providing the semantics of SPAseq using the aforementioned operators. During this process, we review the issues with such an approach and motivate the requirements and behaviour of the newly introduced operators. This results in clearer semantics that can handle all the possible cases. The two main operations that motivate the requirement of introducing new ones are the evaluation of simple GPM expression and the negation over a stream. Note that, here we prefixed the definitions with "Rough-Work", since the readers can confuse them with the correct ones.

From Definition 6.6 in Chapter 6, we know that the evaluation of a graph pattern P over a stream $\mathcal{S}_g$ is as follow:

$$[\![P]\!]_{\mathcal{S}_g} = \{(\tau, [\![P]\!]_G) \mid (\tau, G) \in \mathcal{S}_g\}$$

Based on this, the evaluation of a GPM expression (u, P) over a streamset Σ can roughly be defined as follows:

Rough-Work Definition 8.1: Evaluation of GPM Expression

Given a GPM expression (u, P) *and a streamset* Σ, $[\![(u, P)]\!]_\Sigma$ *can roughly be defined as follows:*

$$[\![(u, P)]\!]_\Sigma = \{(\tau, [\![P]\!]_G) \mid \exists \tau (\tau, G) \in \mathcal{S}_g \wedge (u, \mathcal{S}_g) \in \Sigma\}$$

From Rough-Work Definition 8.1, the evaluation of a GPM expression returns the associated timestamp of an event, and either a set of mappings from the matched events or an empty set in case there is no match with the events. For instance, consider a GPM expression $(u_1, P_1) := (u_1, \{(?h, pow, ?p), (?h, loc, ?l)\})$ and a power-related event $(\tau_i, G_i) := (10, \{(H1,\ pow,\ Pw1), (H15,\ loc,\ L1)\})$ from named stream $(u_1, \mathcal{S}_{g_1}) \in \Sigma$. Then $[\![(u_1, P_1)]\!]_\Sigma = (10, \varnothing)$ for such an event, as the mappings of the variable $?h$ in (τ_i, G_i) are not matched with the graph pattern P_1.

Now consider the evaluation of the negation operator, where a GPM expression is employed to check the non-existence of a certain event. Based on Rough-Work Definition 8.1, we define it roughly as follows.

Rough-Work Definition 8.2: Evaluation of Negation Operator

Given a GPM expression (u, P) *and a streamset* Σ*, the evaluation of the negation operator can roughly be defined as follows:*

$$[\![(u, P)!]\!]_\Sigma = \{(\tau, \varnothing) \mid \exists (u, \mathcal{S}_g) \in \Sigma \wedge\ \forall \tau\ (\tau, G) \in \mathcal{S}_g,\ [\![P]\!]_G = \varnothing\}$$

Thus, for each event that does not match with the graph pattern (P), the evaluation function returns an empty set associated with a timestamp. For instance, using the same GPM expression (u_1, P_1) and an event (τ_i, G_i) from above, evaluation of the negation operator results in an empty set ($[\![(u_1, P_1)\,!\,]\!]_\Sigma = (\tau, \varnothing)$).

Independently, both Rough-Work Definitions 8.1 and 8.2 work fine. However, discrepancies arise when we use them within a sequence expression. Before presenting such an issue, we first define the *followed-by* operator for the sequence between two or more GPM expressions.

Rough-Work Definition 8.3: Evaluation of *Followed-by* Operator

Given two GPM expressions (u_1, P_1) *and* (u_2, P_2)*, the evaluation of the* followed-by *sequence operator over a streamset* Σ *can roughly be defined as follows:*

$$[\![(u_1, P_1); (u_2, P_2)]\!]_\Sigma \quad = \quad \left\{ \begin{array}{c} (\tau', X \bowtie Y) \ \mid\ \exists \tau \tau', (\tau, X) \in [\![(u_1, P_1)]\!]_\Sigma \\ \wedge (\tau', Y) \in [\![(u_2, P_2)]\!]_\Sigma \wedge \tau < \tau' \end{array} \right\}$$

According to Rough-Work Definition 8.3, the evaluation of the *followed-by* operator between two GPM expressions requires the join between the mappings of the preceding GPM expression (or a set of them) and the proceeding one, while considering the total temporal ordering between the matched events. Now we would like to know how the Rough-Work Definitions 8.1 and 8.2 behave according to the definition of *followed-by*

sequence evaluation. We categorise them into two cases as follows:

$$\textit{Case 1}: \quad [\![(u, P_1); (v, P_2)]\!]_\Sigma$$
$$\textit{Case 2}: \quad [\![(u, P_1); (v, P_2)\ !\ ]\!]_\Sigma$$

Case 1 employs simple GPM expressions within a sequence, while *Case 2* uses negation operator within a sequence. Based on these cases, let us consider the following examples to highlight the issues with Rough-Work Definitions 8.2 and 8.3.

Example 12 *Consider two events (τ_i, G_i) and (τ_j, G_j) such that the evaluation of graph patterns P_1 and P_2 over such events are as follows:*

$$[\![P_1]\!]_{G_i} = \Omega, \quad [\![P_2]\!]_{G_j} = \varnothing,$$

where Ω is a set of mappings. Now let us compute both Case 1 and Case 2 for such values of $[\![P_1]\!]_{G_i}$ and $[\![P_2]\!]_{G_j}$. For Case 1, we get $(\tau_j, \varnothing)$, since $\Omega \bowtie \varnothing = \varnothing$, while for Case 2 we get the same $(\tau_j, \varnothing)$ as a result.

According to Example 12, the result of Case 1 is intuitively correct, since the join with an empty set is supposed to be empty mapping. However, intuitively, we were not expecting such a result for Case 2. That is, $[\![P_2]\!]_{G_j} = \varnothing$ means that the negation operator should indicate the non-occurrence of such an event and return the previously matched mappings in the sequence. Thus, we need to define a new structure such that we can differentiate between the empty set from the evaluation of GPM expressions and GPM expressions with negation operator. That is, contrary to the natural join of mappings with empty set $\varnothing \bowtie \Omega = \Omega \bowtie \varnothing = \varnothing$, for the negation operator we need a structure such that $? \bowtie \Omega = \Omega \bowtie ? = \Omega$.

Hence, to describe the behaviour of the negation operator, and to streamline the behaviour of other operators within a sequence, we define an identity element ($\blacktriangle$) for a commutative monoid. It is defined as follows:

Definition 8.5: Identity Element

Given a set of mappings (Ω), where each mapping is denoted as μ. A commutative monoid for mapping set with an identity $\blacktriangle$ element is defined as follows:

$$(2^\Omega \cup \{\blacktriangle\}, \ \tilde{\bowtie}), \textit{ where}$$

- $\forall \mu_1, \mu_2 \in 2^\Omega,\ \mu_1 \tilde{\bowtie} \mu_2 = \mu_1 \bowtie \mu_2$
- $\forall \mu \in 2^\Omega \cup \{\blacktriangle\},\ \mu \tilde{\bowtie} \blacktriangle = \blacktriangle \tilde{\bowtie} \mu = \mu,\ \{\varnothing\} \bowtie \blacktriangle = \blacktriangle \bowtie \{\varnothing\} = \{\varnothing\}$

An identity element permits us to distinguish between the standard empty set of mappings and the empty set produced from the negation operator. Hence, the previous computed mappings within a sequence are not affected by the negation operator.

In this section, we presented the preliminary discussion about some of the issues that can arise while defining the semantics of SPAseq. Motivated by this, we present the semantics of SPAseq operators in the proceeding section.

8.6.2 Semantics of SPAseq Operators

Based on the intuitions from the previous section, for completeness, we define the semantics of SPAseq in a bottom-up manner, where we start with the semantics of graph pattern P by integrating the temporal aspects of the events and streams.

Evaluation of Graph Pattern Matching Expressions

We reuse the definition of graph pattern evaluation from Chapter 6 and extend it for the streamset. Moreover, in order to constrain the evaluation function within a temporal boundary, we use a start time (τ_b) and end time (τ_e) to define the time boundaries, noted $[\tau_b, \tau_e]$. In addition, for the sake of clarity, we use a function $\Sigma(u)$ to select a stream of name u from a streamset, such that

$$\Sigma(u) = \begin{cases} \varnothing & \textit{if } u \textit{ is not a stream name in } \Sigma \\ \mathcal{S}_g & \textit{if } \ (u, \mathcal{S}_g) \in \Sigma \end{cases}$$

Moreover, we denote I as a set of stream names within a streamset Σ. The evaluation of the GPM expression is defined as follows.

Definition 8.6: Evaluation of GPM Expression

*The **evaluation of a GPM** (u, P) over the named stream $(u', \mathcal{S}_g)$, and the streamset Σ is:*

$$\begin{aligned} [\![(u,P)]\!]_{(u',\mathcal{S}_g)} &= \begin{cases} \varnothing & \textit{if } u \neq u' \\ [\![P]\!]_{\mathcal{S}_g} & \textit{otherwise} \end{cases} \\ [\![(u,P)]\!]_{\Sigma}^{[\tau_b,\tau_e]} &= \{(\tau, [\![P]\!]_G) \mid \exists \tau(\tau, G) \in \Sigma(u) \wedge \tau_b \leq \tau \leq \tau_e)\} \end{aligned}$$

The evaluation of the GPM expression (u, P) over an event within a streamset results in a set of mappings annotated with the timestamp of the event. The evaluation of a GPM expression is similar to the semantics of the `GRAPH` construct in SPARQL, where the IRI of the graph is used to select the set of triples to be evaluated for a graph pattern (P).

Example 13 *Consider a GPM expression* $(u_1, P_1) := (u_1, \{(?h, pow, ?p), (?h, loc, ?l)\})$ *and a power-related named stream* $(u_1, \mathcal{S}_{g_1}) \in \Sigma$ *with events as follows:*

$$\mathcal{S}_{g_1} = \{(10, \{(H1,\ pow,\ Pw1),\ (H1,\ loc,\ L1)\}), (15, \{(H2,\ pow,\ Pw2),\ (H2,\ loc,\ L5)\})\}$$

The evaluation of (u_1, P_1) *over* Σ *for the time boundaries [5,10] is described as follows:*

$$[\![(u_1, P_1)]\!]_{\Sigma}^{[5,10]} = (10, \{\{?h \mapsto H1\}, \{?p \mapsto Pw1\}, \{?h \mapsto H1\}, \{?l \mapsto L1\}\})$$

Evaluation of Sequence Expressions

Herein, we describe the evaluation of sequence operators within the sequence expression. We first describe the semantics of event selection strategies, and later use them to recursively define the semantics of unary operators. Let σ be a sequence with a set of GPM expressions and binary/unary operators. The evaluation of *followed-by* operator is defined as follows.

Definition 8.7: Evaluation of Followed-by

*Given a sequence σ and a GPM expression (u, P), the **evaluation of the followed-by (;)** sequence operator over a streamset Σ for the time boundaries $[\tau_b, \tau_e]$ is defined as follows:*

$$[\![\sigma;(u,P)]\!]_{\Sigma}^{[\tau_b,\tau_e]} = \left\{ \begin{array}{c} (\tau, X \bowtie [\![P]\!]_G) \mid \exists \tau\ \tau', (\tau, [\![P]\!]_G) \in [\![(u,P)]\!]_{\Sigma}^{[\tau_b,\tau_e]} \wedge \\ (\tau', X) \in [\![\sigma]\!]_{\Sigma}^{[\tau_b,\tau_e]} \wedge \tau' < \tau \wedge X \neq \blacktriangle \end{array} \right\} \cup$$

$$\left\{ \begin{array}{c} (\tau, [\![P]\!]_G) \mid \exists \tau,\ (\tau, [\![P]\!]_G) \in [\![(u,P)]\!]_{\Sigma}^{[\tau_b,\tau_e]} \wedge \forall \tau' \\ \tau' < \tau\ \wedge (\tau', X) \in [\![\sigma]\!]_{\Sigma}^{[\tau_b,\tau_e]} \wedge X = \blacktriangle \end{array} \right\}$$

The above definition presents two cases: with or without the identity element for the evaluation of sequence σ. Thus, if the sequence σ does not contain a negation operator, the evaluation of the *followed-by* operator is simply the join between the mapping sets from σ and the GPM expression. Otherwise, only the mappings of $[\![P]\!]_G$ are considered according to the property of the identity element in Definition 8.5.

Example 14 *Consider the following, a GPM expression* $(u_1, P_1) := (u_1, \{(?h, pow, ?p), (?h, loc, ?l)\})$ *and a power-related named stream* $(u_1, \mathcal{S}_{g_1}) \in \Sigma$ *as follows:*

$$\mathcal{S}_{g_1} = \{(10, \{(H1,\ pow,\ Pw1),\ (H1,\ loc,\ L1)\}), (25, \{(H2,\ pow,\ Pw2),\ (H2,\ loc,\ L5)\})\}$$

A GPM expression $(u_2, P_2) := (u_2, \{(?w, value, ?v), (?w, loc, ?l)\})$ *and a weather-related named stream* $(u_2, \mathcal{S}_{g_2}) \in \Sigma$ *as follows:*

$$\mathcal{S}_{g_2} = \{(20, \{(W1,\ value,\ Vl1), (W1,\ loc,\ L1)\}), (40, \{(W2,\ value,\ Vl2), (W2,\ loc,\ L8)\})\}$$

And finally a power-storage related named stream $(u_3, \mathcal{S}_{g_3}) \in \Sigma$ *as follows:*

$$\mathcal{S}_{g_3} = \{(15, \{(Pw1,\ status,\ off),\ (Pw1,\ loc,\ L1)\}), (30, \{(Pw1,\ status,\ off),\ (Pw1,\ loc,\ L1)\})\}$$

Then for the evaluation of the followed-by *operator on these GPM expressions for the time boundaries [10,20] we have,*

$$[\![(u_1,P_1);(u_2,P_2)]\!]_{\Sigma}^{[10,25]} = \left\{ \begin{array}{l} (20, \{\{?h \mapsto \mathrm{H1}, ?p \mapsto \mathrm{Pw1}\}, \{?h \mapsto \mathrm{H1}, ?l \mapsto \mathrm{L1}\}, \\ \{?w \mapsto \mathrm{W1}, ?v \mapsto \mathrm{Vl1}\}, \{?w \mapsto \mathrm{W1}, ?l \mapsto \mathrm{L1}\}\}) \end{array} \right\}$$

Notice the mappings of variable ?l from both GPM expressions, since it only matches once (?l = L1) for both power-related and weather-related events as described in the matched results. Furthermore, due to the nature of the followed-by operator, the event from the power-storage related stream $(u_3, \mathcal{S}_{g_3})$ *at* $\tau = 15$ *is skipped between the matched ones at* $\tau = 10$ *and* $\tau = 20$.

We now define the semantics of *immediately followed-by* operator, where I is a set of stream names within a streamset Σ.

Definition 8.8: Evaluation of Immediately Followed-by

*Given a sequence σ and a GPM expression (u, P), the **evaluation of the immediately followed-by (,)** sequence operator over a streamset Σ for the time boundaries $[\tau_b, \tau_e]$ is defined as follows:*

$$[\![\sigma, (u,P)]\!]_{\Sigma}^{[\tau_b,\tau_e]} = \left\{ \begin{array}{c} (\tau, X \bowtie [\![P]\!]_G) \mid \exists \tau\, \tau', (\tau, [\![P]\!]_G) \in [\![(u,P)]\!]_{\Sigma}^{[\tau_b,\tau_e]} \wedge \\ (\tau', X) \in [\![\sigma]\!]_{\Sigma}^{[\tau_b,\tau_e]} \wedge \tau' < \tau \ \wedge \forall \tau'' \, \forall G'' \, \forall i \in I, \\ (\tau'', G'') \in \Sigma(i) \wedge \tau' \leq \tau'', \tau \leq \tau'' \wedge X \neq \blacktriangle \end{array} \right\} \cup$$

$$\left\{ \begin{array}{c} (\tau, [\![P]\!]_G) \mid \exists \tau, (\tau, [\![P]\!]_G) \in [\![(u,P)]\!]_{\Sigma}^{[\tau_b,\tau_e]} \wedge \\ \forall \tau' (\tau', X) \in [\![\sigma]\!]_{\Sigma}^{[\tau_b,\tau_e]} \wedge \tau' < \tau \wedge \forall \tau'' \forall G'' \forall i \in I, \\ (\tau'', G'') \in \Sigma(i) \wedge \tau' \leq \tau'', \tau \leq \tau'' \wedge X = \blacktriangle \end{array} \right\}$$

The semantics of the *immediately followed-by* operator follows the semantics of the *followed-by* operator, however with one important difference: the contiguity between the matched events. That is, an event is *immediately followed-by* another, only if there can be no other events between the two selected ones.

Example 15 *Consider the GPM expressions and the named streams defined in Example 14. Then the evaluation of the* immediately followed-by *over them for time boundaries [10,20] will results in an empty set.*

$$[\![(u_1, P_1);(u_2, P_2)]\!]_{\Sigma}^{[10,20]} = \varnothing$$

This is due to the strict ordering of the immediately followed-by operator. *That is, within the defined window constraints, there is another event at $\tau = 15$, $(\tau, G) = ((15, \{(Pw1, status, off), (Pw1, loc, L1)\}))$ (see Example 14) from the named stream $(u_3, \mathcal{S}_g)$. Even a GPM expression with stream id u_3 is not included in the sequence expression, the strict temporal condition for the* immediately followed-by *operator dictates that there should not be any other event between the two matched ones.*

We now move towards the definition of unary operators, namely negation, optional and kleene-+. We first define their semantics in a standalone manner and then recursively define them with the help of sequence σ.

Definition 8.9: Evaluation of Negation

*The **evaluation of the negation** operator over the streamset Σ with the followed-by and immediately followed-by operator, and for the time boundaries $[\tau_b, \tau_e]$ is defined as follows:*

$$[\![(u,P)!]\!]_{\Sigma}^{[\tau_b,\tau_e]} = \left\{ (\tau, \blacktriangle) \mid (\varnothing, \tau) \in [\![(u,P)]\!]_{\Sigma}^{[\tau_b,\tau_e]} \right\}$$

$$[\![\sigma;(u,P)!]\!]_{\Sigma}^{[\tau_b,\tau_e]} = \left\{ \begin{array}{c} (\tau, X) \in [\![\sigma]\!]_{\Sigma}^{[\tau_b,\tau_e]} \mid \forall \tau' \forall G' \ (\tau', G') \in \Sigma(u) \wedge \\ \tau < \tau' \leq \tau_e \wedge [\![P]\!]_{G'} = \varnothing \end{array} \right\}$$

$$[\![\sigma,(u,P)!]\!]_{\Sigma}^{[\tau_b,\tau_e]} = \left\{ \begin{array}{l} (\tau, X) \in [\![\sigma]\!]_{\Sigma}^{[\tau_b,\tau_e]} \mid \forall \tau' \forall G' \, \forall \tau'' \forall G'' \forall i \in I, \ (\tau', G') \in \Sigma(u) \wedge \\ \tau < \tau' \leq \tau_e \wedge [\![P]\!]_{G'} = \varnothing \wedge (\tau'', G'') \in \Sigma(i) \wedge \tau \leq \tau'', \tau' \leq \tau'' \end{array} \right\}$$

The negation operator determines the non-existence of a certain kind of events and is critical to various SCEP applications. In the aforementioned definition, we use identity element (▲) to track the successful evaluation of the negation operator.

Example 16 *Consider the same GPM expression* (u_1, P_1) *and the named stream* $(u_1, \mathcal{S}_{g_1})$ *from Example 14, and a new GPM expression* $(u_2, P_2) := \{((?w, value, ?v), (?w, loc, ?l)),$ `FILTER` $(?v = Vl3)\}$ *and the same named stream* $(u_2, \mathcal{S}_{g_2})$ *from Example 14. The evaluation of the sequence expression with the negation and followed-by operators for the aforementioned GPM expressions is described as follows:*

$$[\![(u_1, P_1);(u_2, P_2)!]\!]_{\Sigma}^{[10,20]} = \Big\{(20, \{\{?h \mapsto \text{H1}, ?p \mapsto \text{Pw1}\}, \{?h \mapsto \text{H1}, ?l \mapsto \text{L1}\}\})\Big\}$$

Due to the filter expression (`FILTER` *$(?v = Vl3)$), (u_2, P_2) does not match with the events in $(u_2, \mathcal{S}_{g_2})$, and thus produces an identity element according to the semantics of the negation operator. Moreover, observe that the identity element conserves the mappings from the evaluation of* $[\![(u_1, P_1)]\!]_{\Sigma}^{[10,20]}$.

Definition 8.10: Evaluation of Optional

The ***evaluation of the optional*** *operator over the streamset Σ with the followed-by and immediately followed-by operator, and for the time boundaries $[\tau_b, \tau_e]$ is defined as follows:*

$$\begin{aligned}
[\![(u, P)?]\!]_{\Sigma}^{[\tau_b,\tau_e]} &= [\![(u, P)]\!]_{\Sigma}^{[\tau_b,\tau_e]} \cup [\![(u, P)!\,]\!]_{\Sigma}^{[\tau_b,\tau_e]} \\
[\![\sigma;(u, P)?]\!]_{\Sigma}^{[\tau_b,\tau_e]} &= [\![\sigma;(u, P)]\!]_{\Sigma}^{[\tau_b,\tau_e]} \cup [\![\sigma;(u, P)!\,]\!]_{\Sigma}^{[\tau_b,\tau_e]} \\
[\![\sigma,(u, P)?]\!]_{\Sigma}^{[\tau_b,\tau_e]} &= [\![\sigma,(u, P)]\!]_{\Sigma}^{[\tau_b,\tau_e]} \cup [\![\sigma,(u, P)!\,]\!]_{\Sigma}^{[\tau_b,\tau_e]}
\end{aligned}$$

The evaluation of the optional operator is straight-forward from the semantics of GPM expression, negation, *followed-by* and *immediately followed-by* operators.

Example 17 *Consider the GPM expressions* (u_1, P_1) *and* (u_2, P_2) *in Example 14. Let* $((u_1, P_1)$ **;** $(u_2, P_2)?)$ *be the sequence expression, then its evaluation with the optional operator in the presence of named streams* $(u_1, \mathcal{S}_{g_1})$, $(u_2, \mathcal{S}_{g_2}) \in \Sigma$ *(from Example 14) will produce the same set of mappings as illustrated in Example 14. However, if we use the GPM expression* (u_2, P_2) *from Example 16, it will produce the same set of mappings as described in Example 16.*

Definition 8.11: Evaluation of $k+1$

The ***evaluation of*** $k+1$*, where $k > 0$ and general* ***kleene-+*** *operators over the streamset Σ and for the time boundaries $[\tau_b, \tau_e]$ are defined as follows:*

$$\begin{aligned}
[\![(u, P)^1]\!]_{\Sigma}^{[\tau_b,\tau_e]} &= [\![(u, P)]\!]_{\Sigma}^{[\tau_b,\tau_e]} \\
[\![(u, P)^{k+1}]\!]_{\Sigma}^{[\tau_b,\tau_e]} &= [\![(u, P)^k, (u, P)]\!]_{\Sigma}^{[\tau_b,\tau_e]} \\
[\![(u, P)+]\!]_{\Sigma}^{[\tau_b,\tau_e]} &= \bigcup_{k \in \mathbb{N}^*} [\![(u, P)^k]\!]_{\Sigma}^{[\tau_b,\tau_e]}
\end{aligned}$$

The kleene-+ operator groups all the matched events with the defined GPM expression. Note that, we do not illustrate the case of kleene-+ operator with *followed-by* and *immediately followed-by*, since it can easily be inferred from Definitions 8.7 and 8.8.

Example 18 *Consider the following, a GPM expression* $(u_1, P_1) := (u_1, \{(?h, pow, ?p), (?h, loc, ?l)\})$ *and a power-related named stream* $(u_1, \mathcal{S}_{g_1}) \in \Sigma$ *as follows:*

$$\mathcal{S}_{g_1} = \{(10, \{(H1,\ pow,\ Pw1),\ (H1,\ loc,\ L1)\}), (25, \{(H2,\ pow,\ Pw2),\ (H2,\ loc,\ L5)\})\}$$

A GPM expression $(u_2, P_2) := (u_2, \{(?w, value, ?v), (?w, loc, ?l)\})$ *and a weather-related named stream* $(u_2, \mathcal{S}_{g_2}) \in \Sigma$ *as follows:*

$$\mathcal{S}_{g_2} = \{(15, \{(W1,\ value,\ Vl1),(W1,\ loc,\ L1)\}), (20, \{(W1,\ value,\ Vl2),(W1,\ loc,\ L1)\})\}$$

The evaluation of the following sequence with the kleene-+ operator for the time boundaries [10,20] is as follows:

$$[\![(u_1, P_1);(u_2, P_2)+]\!]_{\Sigma}^{[10,20]} = \left\{ \begin{array}{c} (20, \{\{?h \mapsto \text{H1}, ?p \mapsto \text{Pw1}\}, \{?h \mapsto \text{H1}, ?l \mapsto \text{L1}\}, \\ \{?w \mapsto \text{W1}, ?v \mapsto \text{Vl1}\}, \{?w \mapsto \text{W1}, ?l \mapsto \text{L1}\}, \\ \{?w \mapsto \text{W1}, ?v \mapsto \text{Vl2}\}, \{?w \mapsto \text{W1}, ?l \mapsto \text{L1}\}\}) \end{array} \right\}$$

Notice that the kleene-+ operator collects one or more matches for (u_2, P_2) *from the name stream* $(u_2, \mathcal{S}_{g_2})$.

We now move towards the semantics of the binary operators defined for the SPAseq, i.e., conjunction and disjunction of events.

Definition 8.12: Evaluation of Conjunction

Given two GPM expression (u, P) *and* (v, Q)*, the* ***evaluation of the conjunction*** *operator over the streamset* Σ *and for the time boundaries* $[\tau_b, \tau_e]$ *is defined as follows:*

$$[\![(u, P) <> (v, Q)]\!]_{\Sigma}^{[\tau_b, \tau_e]} = \left\{ \begin{array}{c} (\tau, X \bowtie Y) \mid \exists \tau\ (\tau, X) \in [\![(u, P)]\!]_{\Sigma}^{[\tau_b, \tau_e]} \wedge \\ (\tau, Y) \in [\![(v, Q)]\!]_{\Sigma}^{[\tau_b, \tau_e]} \end{array} \right\}$$

The conjunction operator detects the presence of two or more events that match the defined GPM expressions and occur at the same time, i.e., containing the same timestamps.

Example 19 *Consider the following, a GPM expression* $(u_1, P_1) := (u_1, \{(?h, pow, ?p), (?h, loc, ?l)\})$ *and a power-related named stream* $(u_1, \mathcal{S}_{g_1}) \in \Sigma$ *as follows:*

$$\mathcal{S}_{g_1} = \{(10, \{(H1,\ pow,\ Pw1),\ (H1,\ loc,\ L1)\}), (25, \{(H2,\ pow,\ Pw2),\ (H2,\ loc,\ L5)\})\}$$

A GPM expression $(u_2, P_2) := (u_2, \{(?w, value, ?v), (?w, loc, ?l)\})$ *and a weather-related named stream* $(u_2, \mathcal{S}_{g_2}) \in \Sigma$ *as follows:*

$$\mathcal{S}_{g_2} = \{(10, \{(W1,\ value,\ Vl1),(W1,\ loc,\ L1)\}), (20, \{(W1,\ value,\ Vl2),(W1,\ loc,\ L1)\})\}$$

The evaluation of the conjunction operator over these GPM expressions and named streams for the time boundaries [10,20] will results in the following sets of mappings.

$$[\![(u_1, P_1) <> (u_2, P_2)]\!]_{\Sigma}^{[10,20]} = \left\{ \begin{array}{c} (10, \{\{?h \mapsto \text{H1}, ?p \mapsto \text{Pw1}\}, \{?h \mapsto \text{H1}, ?l \mapsto \text{L1}\}, \\ \{?w \mapsto \text{W1}, ?v \mapsto \text{Vl1}\}, \{?w \mapsto \text{W1}, ?l \mapsto \text{L1}\}\}) \end{array} \right\}$$

Definition 8.13: Evaluation of Disjunction

Given two GPM expression (u, P) *and* (v, Q)*, the* ***evaluation of the disjunction*** *operator over the streamset* Σ *and for the time boundaries* $[\tau_b, \tau_e]$ *is defined as follows:*

$$[\![(u, P) \mid (v, Q)]\!]_{\Sigma}^{[\tau_b, \tau_e]} = [\![(u, P)]\!]_{\Sigma}^{[\tau_b, \tau_e]} \cup [\![(v, Q)]\!]_{\Sigma}^{[\tau_b, \tau_e]}$$

The disjunction operator detects the occurrence of events that match to a GPM expression within the set of defined ones.

Example 20 *Consider the following, a GPM expression* $(u_1, P_1) := (u_1, \{(?h, pow, ?p), (?h, loc, ?l)\})$ *and a power-related named stream* $(u_1, \mathcal{S}_{g_1}) \in \Sigma$ *as follows:*

$$\mathcal{S}_{g_1} = \{(22, \{(H1,\ pow,\ Pw1),\ (H1,\ loc,\ L1)\}), (25, \{(H2,\ pow,\ Pw2),\ (H2,\ loc,\ L5)\})\}$$

A GPM expression $(u_2, P_2) := (u_2, \{(?w, value, ?v), (?w, loc, ?l)\})$ *and a weather-related named stream* $(u_2, \mathcal{S}_{g_2}) \in \Sigma$ *as follows:*

$$\mathcal{S}_{g_2} = \{(20, \{(W1,\ value,\ Vl1),(W1,\ loc,\ L1)\}), (40, \{(W2,\ value,\ Vl2),(W1,\ loc,\ L8)\})\}$$

A GPM expression $(u_3, P_3) := (u_3, \{(?p, status, ?s), (?p, loc, ?l)\})$ *and a power-storage related named stream* $(u_3, \mathcal{S}_{g_3}) \in \Sigma$ *as follows:*

$$\mathcal{S}_{g_3} = \{(15, \{(Pw1,\ status,\ on),\ (PW1,\ loc,\ L1)\}), (30, \{(Pw1,\ status,\ off),\ (PW1,\ loc,\ L1)\})\}$$

The evaluation of the disjunction sequence operator with the followed-by *operator for the following sequence expression, and for the time boundaries [10,20] is as follows:*

$$[\![((u_1, P_1) \mid (u_3, P_3)) ; (u_2, P_3)]\!]_{\Sigma}^{[10,20]} = \left\{ \begin{array}{c} (20, \{\{?p \mapsto \mathrm{Pw1}, ?s \mapsto \mathrm{on}\}, \{?p \mapsto \mathrm{Pw1}, ?l \mapsto \mathrm{L1}\}, \\ \{?w \mapsto \mathrm{W1}, ?v \mapsto \mathrm{Vl1}\}, \{?w \mapsto \mathrm{W1}, ?l \mapsto \mathrm{L1}\}\}) \end{array} \right\}$$

8.6.3 Evaluation of SPAseq Queries

In the previous section, we outline the semantics of main temporal operators of SPAseq. Herein, to sum it up, we present the evaluation semantics of complete SPAseq queries.

Let Ω be a mapping set and $\pi_{\mathcal{V}}$ be the standard SPARQL projection on the set of variables $\mathcal{V}$, ω be the duration of the window, then the evaluation of SPAseq query $Q = (\mathcal{V}, \omega, \mathrm{SeqExp})$ issued at time t, over the streamset Σ is defined as follows:

Definition 8.14: Evaluation of SPAseq Query

$$[\![Q]\!]_{\Sigma}^{t} = \bigcup_{k \in \mathbb{N}} \left\{ (\tau, \pi_{\mathcal{V}}(\Omega)) \mid (\tau, \Omega) \in [\![\mathrm{SeqExp}]\!]_{\Sigma}^{[t+k\cdot\omega, t+(k+1)\cdot\omega]} \right\}$$

$$\textit{where} \quad \pi_{\mathcal{V}}(\Omega) = \{\mu_1 \mid \exists \mu_2 : \mu_1 \cup \mu_2 \in \Omega \wedge dom(\mu_1) \subseteq \mathcal{V} \wedge dom(\mu_2) \cap \mathcal{V} = \varnothing\}$$

The evaluation of the SPAseq queries follow a push-based semantics, i.e., results are produced as soon as the sequence expression matches to the set of events within the streamset. Thus, the resulting set of mappings takes the shape of a stream of

mappings, where the order within the mappings depends on the underlying executional framework. Note that the definition of $[\![Q]\!]^t_\Sigma$ is the intended one. It could be possible to define a continuous version of the query evaluation but we want to stay agnostic to how the solutions are provided. For instance, the evaluation could be performed on a static file with time series, possibly including future previsions; or the solutions could be provided in bulks every ω time units.

Example 21 *Recall the two GPM expressions from Example 14,* $(u_1, P_1) := (u_1, \{(?h, pow, ?p), (?h, loc, ?l)\})$ *and* $(u_2, P_2) := (u_2, \{(?w, value, ?v), (?w, loc, ?l)\})$. *Now consider the two named streams* $(u_1, \mathcal{S}_{g_1})$, $(u_2, \mathcal{S}_{g_2}) \in \Sigma$ *as follows:*

$$\mathcal{S}_{g_1} = \{(10, \{(H1,\ pow,\ Pw1),\ (H1,\ loc,\ L1)\}), (25, \{(H2,\ pow,\ Pw2),\ (H2,\ loc,\ L5)\})\}$$

$$\mathcal{S}_{g_2} = \{(15, \{(W1,\ value,\ Vl1), (W1,\ loc,\ L1)\}), (40, \{(W2,\ value,\ Vl2), (W1,\ loc,\ L8)\})\}$$

The evaluation of a SPAseq *query* $Q = (\{?h, ?p, ?v\}, 50, ((u_1, P_1)\ ;\ (u_2, P_2)))$ *at time* $\tau = 20$ *over the streamset* Σ *can be described as follows:*

$$[\![Q]\!]^{20}_\Sigma = \Big\{(10, \{\{?h \mapsto H1\}, \{?p \mapsto Pw1\}), (15, \{\{?v \mapsto Vl1\}\})\Big\}$$

8.7 Qualitative Comparative Analysis

In this section, we present the qualitative comparison between SPAseq and EP-SPARQL. While a complete formal comparison between both is certainly very interesting, we will leave it for future work and, instead, focus on a use case-based comparison of these two languages.

8.7.1 Input Data Model

As discussed in Chapter 4 and Chapter 5, RSP and SCEP systems are evolved from DSMSs and CEP systems respectively. Thus, the mapping of triples to tuples seems to be the obvious choice for existing SCEP systems, leading to a triple stream model. However, events (within relational data model) do not consist of individual data items but rather a set of them. The decomposition of data items within an event into a set of RDF triples for triple streams cannot directly represent the boundaries of data items within events, and a query that observers only a partial event may return false results. Moreover, in order to support heterogeneous streams, the system must be able to handle streams for which neither the interval between events, nor the the number of triples in an event is known in advance. Hence, streaming a set of RDF triples together as an event would not only greatly simplify the task for event producers, since neither the order of decomposition of event object graphs, nor the addition of triples needs to be considered, but it can also increase the performance of the system (as described in Chapter 7 (Section 7.7.2)).

In the following, we use a simple example to show case differences between EP-SPARQL and SPAseq based on their data model: EP-SPARQL uses a triple event model, while SPAseq employs an RDF graph-based event model.

Consider a simple form of a trajectory detection use case (as described in **UC 3**), where a user is interested in finding the speed of a boat for sequence: if it is directed towards "`south`" followed-by a direction towards "`north`". Now consider that a boat is

equipped with a sensor describing the values of the current direction and the speed of the boat. The streams generated from the boat's sensor for both RDF event stream (S_{re}) and RDF graph event stream ($\mathcal{S}_g$) models are described as follows:

$$S_{re} = \begin{Bmatrix} \{\langle boat1, direction, south\rangle, [\tau_1, \tau_{1+1}]\}, \\ \{\langle boat1, speed, 60\rangle, [\tau_2, \tau_{2+1}]\}, \\ \{\langle boat1, direction, north\rangle, [\tau_3, \tau_{3+1}]\}, \\ \{\langle boat1, speed, 70\rangle, [\tau_4, \tau_{4+1}]\}, \ldots \end{Bmatrix}$$

$$\mathcal{S}_g = \begin{Bmatrix} \{\langle boat1, direction, south\rangle, \{\langle boat1, speed, 60\rangle, \tau_1\}, \\ \{\langle boat1, direction, north\rangle, \{\langle boat1, speed, 70\rangle, \tau_2\}, \ldots \end{Bmatrix}$$

Based on the above mentioned streams, intuitively, we can define EP-SPARQL (Query 8.6) and SPAseq (Query 8.8) queries for the case described above.

```
SELECT ?s1 ?s2
      WHERE

            SEQ { ?boat :direction ?d1.
                  ?boat :speed ?s1.
                }

            SEQ { ?boat :direction ?d2.
                  ?boat :speed ?s2.
                }

Filter (?d1 = "south" && ?d2= "north" && getDURATION() < "P30M"^^
   xsd:duration))
```

Query 8.6: EP-SPARQL Query with `SEQ` Clause

```
SELECT ?s1 ?s2
      WHERE

            SEQ { ?boat :direction ?d1.
                  EQUALS {
                  ?boat :speed ?s1.
                         }
                }

            SEQ { ?boat :direction ?d2.
                  EQUALS {
                  ?boat :speed ?s2.
                         }
                }

Filter (?d1 = "south" && ?d2= "north" && getDURATION() < "P30M"^^
   xsd:duration))
```

Query 8.7: EP-SPARQL Query with `SEQ` and `EQUALS` Clauses

According to the Query 8.6 and semantics of EP-SPARQL, there will be a match over RDF event stream if: (i) two consecutive triples with the same time-intervals having

mapping of `?d1 = 'south'`, and any mappings of `?s1` *followed-by* (ii) two consecutive triples with the same time-intervals having mapping of `?d2 = 'north'` arrive within an RDF event stream. However, according to the stream model, such a query cannot be matched with S_{re}. Therefore, the user has to perform a couple of tasks to evaluate the use case for an EP-SPARQL query: (i) change the structure of the stream to have the same time-intervals for triples from two different sources, or make sure that the order between the triples remains the same (not practical in real-world situations), (ii) write a complex query with the combination of `EQUAL` and `SEQ` operators (see Query 8.7). This questions the usability and performance of EP-SPARQL queries.

Now consider the SPAseq Query 8.8 defined for the stream $\mathcal{S}_g$. Due to the RDF graph-based stream model, and the separation of GPM and sequence expressions, SPAseq not only provides an intuitive way of writing queries, but also complies to the real-world situations. That is, a set of triples expressing the attributes of a source associated with a time-stamp. This means, the source does not have to comply to a certain order when producing the triples, instead the produced triples can be packaged into a single graph. The execution of SPAseq Query 8.8 over $\mathcal{S}_g$ will produce the required matches, i.e., the selection of mappings of `?s1` and `?s2` $((\tau_2, \{\{?s1 \mapsto 60\}, \{?s2 \mapsto 70\}\}))$.

```
1
2  SELECT ?s1 ?s2
3  WITHIN 30 MINUTES
4  FROM STREAM S1 <http://harbour.org/boats>
5
6   WHERE {
7
8      SEQ (A; B)
9      DEFINE GPM A ON S1 {
10      ?boat :direction ?d1.
11      ?boat :speed  ?s1.
12       Filter (?d1 = 'south')
13     }
14        DEFINE GPM B ON S1 {
15        ?boat :direction ?d2.
16        ?boat :speed  ?s2.
17         Filter (?d2 = 'north')
18        }
19
20   }
21
```

Query 8.8: Trajectory Classification: SPAseq query

Another obvious difference between the data model of EP-SPARQL and SPAseq is the streamset: SPAseq queries are evaluated on a streamset where a set of heterogeneous streams can be used, while EP-SPARQL queries are evaluated on a single stream. For the same reason, semantically it is not possible to support **UC 1** with the EP-SPARQL queries.

8.7.2 TimePoints Vs Time-Intervals

As evident from the data models, SPAseq temporal semantics is based on points in time, while EP-SPARQL utilises time-intervals. The choice of the timepoints for the SPAseq is based on the following reasons.

1. W2C RSP working group (as discussed earlier) has been working on the standardising of RDF stream model for the last three years or so. Recently, they have provided a draft version of their recommendations[2], where timepoint-based semantics for the

[2]RDF Stream Abstract Syntax and Semantics: `http://streamreasoning.github.io/RSP-QL/Abstract%20Syntax%20and%20Semantics%20Document/` , last accessed: July, 2016.

RDF streams are recommended.

2. Due to the complexity of the RDF data model, there are obvious and considerable performance differences between relational CEP and SCEP systems. Our aim of providing a new SCEP language and system is to close such gaps, while providing expressive CEP operators over the RDF data model. Timepoint-based semantics perfectly fit in this context and most of the performance intensive CEP systems rely on it, such as SASE [WDR06b, Agr+08], Esper [BV10].

3. Existing CEP and SCEP systems are based on single stream model, and multiple streams have not gained much attention. SPAseq provides temporal operators over a streamset, and thus we have consulted various DSMSs and RSP systems to weigh up the differences between timepoints and time-intervals. In the case of time-intervals, the implementation of joins between different streams over windows is not a straight-forward task and requires careful considerations: Kraemer *et al.* have examined such issues in detail for the DSMSs [HV05].

The use of timepoints results in a cleaner semantics with the focus on how the RDF graph-based events and temporal operators are evaluated in an optimised manner. Although the time-interval based temporal model offers associativity of sequence operator, it can be considered as an extension of our system to handle events with duration. One of such technique is called *coalescing* from the temporal database [BSS96]. Coalescing is a unary operator for merging value-equivalent elements with adjacent time intervals in order to build larger time-intervals. For instance, consider two fact-based temporal triples (`person1, inside-room, r1, [15:00]`) and (`person1, inside-room, r1, [18:00]`), where the two temporal triples can be replaced with a single one with a time-interval (`person1, inside-room, r1, [15:00,18:00]`). The coalescing operator can be applied over the evaluation of temporal operators to extract the intervals over the matching set of mappings. Moreover, the timestamp in each RDF graph event can be mapped to time-intervals, i.e., and an event $(\tau, G) \in \mathcal{S}$ to $([\tau, \tau + 1], G)$, and for the primitive events start and end timestamps can be the same. This would not affect our semantics, since time-interval $[\tau, \tau + k]$ solely covers a single time, namely τ [HV05].

8.7.3 Temporal Operators

In our previous discussion, we have emphasised on the clear differences between the supported temporal operators for EP-SPARQL and SPAseq. Herein, we focus on the kleene-+ operator and show its importance.

Recall the V-shaped pattern from **UC 2**. EP-SPARQL, unlike SPAseq, it only supports the simplified V-shaped pattern using the `SEQ` and `FILTER` operators. However, complex situations with a kleene-+ operator are not supported. We illustrate this through an example scenario. Figure 8.2b(a) shows a strict V-shaped pattern, where the events should follow the strict sequence ($e_2.value < e_1.value$ **followed-by** $e_3.value > e_2.value$ **followed-by** $e_3.value > e_1.value$). This strict sequence is supported by both query languages. However, a relaxed sequence pattern as described in Figure 8.2b(b) requires a *kleene-+* operator to consume one or more events of the same kind, and is not supported by EP-SPARQL. The kleene-+ operator is widely used in diverse domains as discussed earlier, especially for sensor networks.

In Section 8.4, we discuss the two event selection operators, *immediately followed-by* and *followed-by* operators. The semantic analysis of EP-SPARQL (as presented in

Figure 8.2: V-Shaped Patterns (a) Without Kleene-+ Operator, and (b) With Kleene-+ Operator

Chapter 5) shows that the general sequence operator of EP-SPARQL closely corresponds to the *immediately followed-by* operator. However, the *followed-by* operator, with wide spectrum of use cases, is not supported by the EP-SPARQL query language.

8.8 Summary

In this chapter, we have answered the question *How to define a new SCEP query language and which operators should it use?*. We have presented the syntax and semantics of SPAseq, a SCEP query language. We have also provided the motivations behind our language, and pointed out various qualitative differences between SPAseq and another SCEP query language EP-SPARQL. The contributions of our work in this chapter are the following:

- **Streaming Data Model:** We have proposed a new streaming data model, i.e., RDF graph-based events, where each event within a stream is a graph annotated by a timestamp. These RDF graph-based events are observed in streams that are also named. Moreover, we have also proposed the notion of streamset to evaluated SCEP queries over multiple heterogeneous RDF graph streams.
- **Syntax of SPAseq:** We have defined the syntax of the SPAseq language, where GPM expressions are separated from the sequence expressions. This enables expressive temporal operators over the streamset.
- **Semantics of SPAseq:** Based on the defined syntax, we have also provided the evaluation semantics of temporal operators introduced in SPAseq. That is, how a SPAseq query is evaluated and its expected results.
- **Qualitative Comparison:** Motivated by the real-world use cases, we have provided a non-formal qualitative comparison of SPAseq and its competitor EP-SPARQL. Our analysis showed that, SPAseq improves upon EP-SPARQL in many aspects, such as data model and temporal operators.

The design of the syntax and semantics of the SPAseq query language can guide the practitioners to compare their works, and to extend the SCEP languages with further properties. During the design phase of our language, we carefully consulted existing CEP techniques and the lessons learned. Thus, a suitable compromise between the expressiveness of a SCEP language and how it can be implemented in an effective way is made possible. In the following chapter, we will ponder on the optimised implementation of SPAseq and its operators.

In dancing, a single step, a single movement of the body that is graceful and not forced, reveals at once the skill of the dancer. A singer who utters a single word ending in a group of four notes with a sweet cadence, and with such facility that he/she appears to do it quite by chance, shows with that touch alone that he/she can do much more than he is doing.

— Baldassare Castiglione, *The Book of the Courtier*

9

SPAseq: Semantic Complex Event Processing over RDF Graph Streams

In the previous chapter, we discussed the design of a new SCEP language called SPAseq*. This chapter provides the implementation details of* SPAseq*, i.e., how its operators are compiled and executed in an efficient way. The underlying execution model of* SPAseq *is based on NFA_{scep}, where a set of states, each with a set of edges, are used to map* SPAseq *operators. We first provide the motivation of using the NFA_{scep} model, and then provide the details of its compilation process for the* SPAseq *query language. Later, we present how an NFA_{scep} automaton is executed, its evaluation complexity and various optimisation strategies customised for SCEP.*

Contents

This chapter is structured as follows: Section 9.1 provides the preliminary introduction and recalls the key concepts of CEP systems. Section 9.2 presents the

NFA_{scep} model and algorithms to compile SPASEQ *operators to an equivalent NFA_{scep}. Section 9.3 details our system design and algorithms for evaluating* SPASEQ *queries. Section 9.4 presents a list of optimisation techniques utilised by the* SPASEQ *query engine. Section 9.5 presents the experimental evaluation of* SPASEQ *queries. Section 9.6 concludes the chapter.*

9.1 General Idea

Pattern matching techniques have been employed in most of the branches of science. In general, pattern matching aims to obtain, or at-least, access the correlation between two sets of data. Its concluding aim is to either get a simple binary *yes* or *no* answer, or to determine a set of independent parameters that produce the best match between the two data sets. In the context of CEP, pattern matching is to match a set of temporal operators, defined in a query language, with a set of events within an event stream.

Hence, with the arrival of events, defined temporal operators are evaluated in a progressive way. That is, before a composite or complex event is detected (please refer to Chapter 5 to refresh such concepts) through a full pattern match, partial matches of the query patterns emerge with time. These partial matches require to be taken into account, primarily within in-memory caches, since they express the potential for an imminent full match. As discussed in Chapter 5, there exists a wide spectrum of approaches to track the state of partial matches, and to determine the occurrence of a complex event. In summary, these approaches include rule-based techniques that mostly represent a set of rules in tree structures (such as RETE network), graph-based representations (such as Event Detection Graphs (EDGs)) to merge all the rules within a single structure, and finally Finite State Machine representations, in particular non-deterministic Finite Automata (NFA). The choice of these representations is motivated not only by their expressiveness measures, but also on the performance metrics that each approach tries to enhance. For instance, ETALIS [Ani+12], a rule-based engine, mostly focuses on how the complex rules are mapped and executed as Prolog objects and rules, while SASE [WDR06b, Agr+08] and Zstream [MM09] focus on the query-rewriting, predicate-related optimisations and memory management techniques. Table 9.1 illustrates some of the optimisation strategies utilised by the CEP systems[1], and their description is provided as follows:

- *Query-rewriting.* This technique, as described in Chapter 3, takes its basis from the DBMSs and DSMSs. Its aim is to re-order query operators, either offline using their selectivity measures, or online using statistical information from the event stream. This leads to low-cost selective operators to be evaluated first, and thus reducing the intermediate results and load on the remaining operators.

- *Predicate-based Optimisations.* Predicates within a CEP query present the constraints and filter the event data. A simple but efficient technique for CEP optimisation is to push the predicates as early as possible in query plans. This reduces the number of partial matches by eliminating the ones that would not result in a complete match.

- *Memory Management.* CEP over unbounded stream is expensive in terms of memory usage and the partial matches can grow exponentially [Agr+08, MM09]. The aim of the memory-based optimisation is to share the results from partial matches and

[1]We only presents the well-know research systems that describe their design in details.

merge them to reduce the memory usage. One such technique [Agr+08] defines buffers for each partial match and then merges individual buffers into a single one.

Table 9.1: Available Optimisation Strategies Adopted by the CEP Systems

CEP Systems	Query-rewriting	Predicate-based Optimisations	Memory Management
SASE [Agr+08]	✓	✓	✓
ETALIS [Ani+12]	✗	✗	✗
Cayuga [Bre+07]	✓	✗	✗
Zstream [MM09]	✓	✓	✗

The above discussion highlights some of the areas that could be explored to optimise CEP. However, in this thesis, we are concerned with the SCEP: with the integration of RDF data model, the story becomes complicated, since SCEP not only requires the efficient management of temporal operators, but also the efficient evaluation of graph patterns. Therefore, our efforts for an efficient implementation of SCEP are concentrated on customising some of the existing CEP-based techniques – where SASE is the best source of inspiration – along-with some new insights and integration of the RDF data model.

9.2 NFA-based Semantic Complex Event Processing

Pattern matching is commonly performed by expressing patterns as sets of regular expressions and by converting them into finite state automata (FSAs). The behaviour of FSA is easy to emulate on computing devices to perform matching, and FSA can easily be composed together with a full set of boolean operators. Two different kinds of automata models that are proposed in the literature include: deterministic finite automata (DFA) and its non-deterministic version NFA. While theoretically, both models possess the same expressive power, DFA can be less space efficient, requiring very large amount of memory to store, and can result in state space explosions [BC08].

The design choice of temporal operators in SPAseq is heavily influenced by whether they can be efficiently evaluated or not. Our criterion for the efficiency of a SPAseq temporal operator is whether it can be mapped to an NFA. The rational behind choosing NFA as the underlying execution model is two fold. First, SPAseq is designed for complex temporal patterns and these patterns can be intuitively described as transitions in an automaton: NFAs are expressive enough to capture all the complex patterns in SPAseq. Second, NFAs retain many attractive computational properties of FSA on words: by translating SPAseq queries into NFAs, we can exploit several existing optimisation techniques [Agr+08, ZDI14]. Table 9.1 illustrates some of the optimisation techniques for CEP systems. The SASE [Agr+08] and Cayuga [Bre+07], where both employ the NFA model, are the dominating ones with a number of custom optimisations.

In the following, first we describe the execution model of our system, and later present how SPAseq queries are complied onto equivalent NFAs. A discussion on the optimisation techniques for the execution of SPAseq queries is provided in Section 9.4.

9.2.1 NFA$_{scep}$ Model for SPAseq

We designed a new type of automata model, called as NFA$_{scep}$, where the GPM expressions are mapped as state-transition predicates for the automaton states. Formally, an NFA$_{scep}$

automaton is defined as follows.

Definition 9.1: NFA$_{scep}$ Automaton

An NFA$_{scep}$ automaton is a tuple $\mathcal{A} = \langle X, E, \Theta, \varphi, x_o, X_m, x_f \rangle$, where

X: is a set of states;

E: a set of directed edges connecting states;

Θ: is a set of state-transition predicates, where each $\theta \in \Theta$, $\theta = \{\mathrm{u}, \mathrm{sf}, \mathrm{op}, \mathrm{P}\}$; u is the stream name, sf is the set of statefully-joined triples, $\mathrm{op} \in \{$'?', '!', '<>', '+', '/'$\}$ is a temporal operators, and P is a graph pattern;

φ: is a labelling function $\varphi : E \rightarrow \Theta \cup \{\epsilon\}$ that maps each edge to the corresponding state-transition predicate, where ϵ denotes the instantaneous transition [Tho68];

x_o: $x_o \in X$ is an initial or starting state;

X_m: $X_m \subset X$ is a set of manager states;

x_f: $x_f \in X$ is a final state.

Each NFA$_{scep}$ automaton is acyclic, except for the self loops, and we define four types of states: *initial* (x_o), *ordinary* (x), *final* (x_f) and *manager* (x_m) states. The first three types (initial, ordinary and final) are analogous to the states in the traditional NFA models in order to implement the basic operators such as sequence, negation, kleene-+. The manager state provides an important functionality to implement disjunction and conjunction operators in an optimised manner (details will follow in Section 9.4). However, semantically it works similarly to that of ordinary state. Each state has at least one forward edge, except the final state.

Example 22 *Figure 9.1 shows the compiled NFA$_{scep}$ for the* SPASEQ *Query 8.2 with the sequence expression* `SEQ(A,B+,C)`*. It contains four states, each having a set of edges labelled with the state-transition predicates. The state-transition predicate (u_s, sf, op, P) consists of four parameters: graph pattern P for the events with stream id (u_s); sf represents the set of stateful joins, for instance, variable $?fr1$ in Query 8.2 is shared between P_A and P_B; op describes the type of operator mapped to an edge, for instance edges of state x_1 contains the kleene-+ operator. The description of mapping from the* SPASEQ *Query 8.2 to the NFA$_{scep}$ in Figure 9.1 is as follows:*

- *The* SPASEQ *Query 8.2 contains the sequence expression* `SEQ(A,B+,C)`*, thus there is one initial state, two ordinary states and a final state.*
- *State x_0 has one edge with state-transition predicate (u_{s_1}, sf_1, $\varnothing$, P_A). Since it only contains* immediately followed-by *operator as a temporal operator, it can simply transit to the next state on matching the state-transition predicate. Note that, the "," represents* immediate followed-by operator*: the case of* followed-by *operator (";") is described later.*

- *The state x_1 represents the mapping of* `GPM B` *with kleene-+ operator. Therefore, it has two edges each with a state-transition predicate (u_{s_2}, sf_2, +, P_B), one with a destination state of x_2, and other with the same destination (x_1) to consume one or more same kind of events.*
- *The state x_2 represents the mapping of* `C`*, hence one edge is used to transit to next state if an event matches the defined state-transition predicate (u_{s_3}, sf_3, $\varnothing$, P_C).*

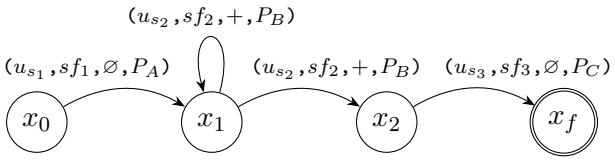

Figure 9.1: Compiled NFA_{scep} for SPAseq Query 8.2 with `SEQ(A,B+,C)` expression

State-transition predicates are used to determine the action taken by a state to transit to another. For instance, in Figure 9.1 the state x_0 transits to x_1, if (1) the incoming event is from the defined stream id u_{s_1}, (2) if there are stateful joins between different graph patterns, and their evaluation do not result in an empty set of mappings, (3) graph pattern P_A evaluates to true. Furthermore, the event selection strategy also determines if there is a *followed-by* or *immediately followed-by* relation between the processed events. Note that, in the presence of kleene-+ operator, NFA_{scep} will exhibit the non-determinism behaviour, since the state-transition predicates will not be mutually exclusive.

Considering the same vocabulary from the existing NFA works (as described in Chapter 5), we say that each instance of an NFA_{scep} is called a *run*. A run depicts the partial matches of defined patterns, and contains the set of selected events. Each run has a current *active state*: a run that reaches its final state is an *accepting run*, denoting that all the defined patterns are matched with the set of selected events.

9.2.2 Compiling SPAseq Queries

As described in Chapter 8, the two main components of the SPAseq language are *sequence* and *GPM expressions*. Due to the separation of these components, one can provide variable techniques to compile and process them. In Chapters 6 and 7, we review the compilation process of graph patterns using the traditional relational operators (e.g., selection, projection, cartesian product, join, etc.) within each GPM expression. Herein, using existing knowledge, we show how stateful joins are handled between a set of GPM expressions, and how these expressions along-with the temporal constraints are mapped onto a set of state-transition predicates: by stateful joins we mean if the variables are shared between two or more GPM expressions. The compiled set of state-transitions predicates is later used to fulfil the labelling of automaton state's edges.

Algorithm 5 shows the compilation process of a set of GPM expressions. First each GPM expression is examined against the remaining set to determine the stateful variables that result in joins at *subjects*, *predicates* and *objects* level (*lines 5 – 15*). The compiled set of stateful joins, and the stream ids, which would be utilised for the evaluation of GPM expressions, are mapped to the corresponding state-transition predicates (*lines 19 –*

21). As previously discussed, the set of state-transition predicates are used by the atomic elements of sequence expression (see Algorithm 6) in order to construct the automaton states, and to label the corresponding edges (as described in Example 22).

Algorithm 5 Compiling GPM expressions within a SPASEQ Query

```
1: gpmSet ← {(u_1, P_1), (u_2, P_2), ..., (u_m, P_m)}
2: Θ ← {}
3: procedure CompilePredicates (gpmSet , Θ)
4: for each (u_i, P_i) ∈ gpmSet do
5:     stateful ← {}
6:     checkJoin ← gpmSet \ {(u_i, P_i)}
7:     for each tp ∈ P_i do
8:         if sub(tp) ∈ V && sub(tp) ∈ checkJoin then
9:             stateful ← stateful ∪ {P_i ⋈_sub(tp) getSub(tp, checkJoin)}
10:        end if
11:        if pred(tp) ∈ V && pred(tp) ∈ checkJoin then
12:            stateful ← stateful ∪ {P_i ⋈_pred(tp) getPred(tp, checkJoin)}
13:        end if
14:        if obj(tp) ∈ V && obj(tp) ∈ checkJoin then
15:            stateful ← stateful ∪ {P_i ⋈_obj(tp) getObj(tp, checkJoin}
16:        end if
17:    end for
18:    θ ← newPredicate()
19:    sID_u(θ) ← u_i
20:    stateful_st(θ) ← stateful
21:    graphPatt_P(θ) ← P_i
22:    Θ ← Θ ∪ θ
23: end for
```

The sequence expression sorts the execution of GPM expressions according to its entries. Moreover, the temporal operators determine the occurrence criteria of such GPM matches, and the event selection strategies are utilised to select the relevant events. These constraints or properties are mapped on the NFA_{scep} through the compiled state-transition predicates, and Algorithm 6 shows such compilation process. For the sake of brevity, we do not consider the conjunction and disjunction operators in Algorithm 6; such process is intuitive enough and will be explained in the proceeding discussion.

Algorithm 6 takes three inputs: (i) a list of *atoms* from the sequence expression, each containing a temporal operator, an event selection strategy, and a GPM expression; (ii) a set of compiled state-transition predicates from Algorithm 5; (iii) a set of states whose edges will be mapped with the corresponding predicates. It starts by iterating over the list of atoms, and gets the state-transition predicate that matches the GPM expression of the atom (*line 5*). It then uses the temporal operator and appends it to the corresponding predicate (*line 6*). Finally, it starts the process of mappings state-transition predicates to the state's edges: it iteratively adds new edges and labels them with state-transition predicates.

For the optional and negation operators ('?', '!'), Algorithm 6 adds one edge labelled with the defined predicate, such that the destination of the edge is the next state (*lines 7–8*), and another edge labelled with ϵ-transition, i.e., transition on the empty word [Tho68] (see Figure 9.4 and 9.6). For the kleene-+ (+) operator there are two edges, each containing the same predicates, while one destined to the next state, and the other comes back to the source state (*lines 11–15*) (see Figure 9.5). The same is the case with the

followed-by operator, where an edge has the same source and destination states. Hence, the irrelevant events ($\neg n$) are skipped (*lines* 16–20) (see Figure 9.3). Let (u_1, P_2) and

Algorithm 6 Compilation of NFA$_{scep}$ states and edges for a SPASEQ Query

1: $Atom \leftarrow \{at_1, at_2, \ldots, at_i\}$, where $at \leftarrow \{(u, P), op, es\}$, $op \leftarrow \{$ '?' , '!', '+'$\}$, $es \leftarrow \{$ ',' , ';'$\}$
2: $\Theta \leftarrow \{\theta_1, \theta_2, \ldots, \theta_n\}$, from Algorithm 5.
3: $\mathcal{A} \leftarrow \{x_1, x_2, \ldots, x_{n+1}\}$, where $\texttt{edges}(x_i) \leftarrow \{\}$, $\texttt{next}(x_i) = x_{i+1}$
4: **procedure** COMPILEEDGES(*Atom*, $\mathcal{A}$, Θ)
5: **for each** $at_i \in Atom$ **do**
6: $\quad \theta \leftarrow \texttt{getMatchPredicate}\ (\Theta, (u, P))$
7: $\quad \texttt{opera}_{op}(\theta) \leftarrow \texttt{opera}_{op}(at_i)$
8: $\quad$ **if** $\texttt{opera}_{op}(\theta) =$ '!' **then**
9: $\quad\quad \texttt{graphPatt}_P(\theta) \leftarrow \neg\texttt{graphPatt}_P(\theta)$
10: $\quad\quad e_1 \leftarrow \texttt{newEdge}(\theta)$, $\texttt{source}(e_1) \leftarrow x_i$, $\texttt{destination}(e_1) \leftarrow x_{i+1}$
11: $\quad\quad e_2 \leftarrow \texttt{newEdge}(\epsilon)$, $\texttt{source}(e_2) \leftarrow x_i$, $\texttt{destination}(e_2) \leftarrow x_{i+1}$
12: $\quad\quad \texttt{edges}(x_i) \leftarrow \texttt{edges}(x_i) \cup \{e_1\} \cup \{e_2\}$
13: $\quad$ **end if**
14: $\quad$ **if** $\texttt{opera}_{op}(\theta) =$ '?' **then**
15: $\quad\quad e_1 \leftarrow \texttt{newEdge}(\theta)$, $\texttt{source}(e_1) \leftarrow x_i$, $\texttt{destination}(e_1) \leftarrow x_{i+1}$
16: $\quad\quad e_2 \leftarrow \texttt{newEdge}(\epsilon)$, $\texttt{source}(e_2) \leftarrow x_i$, $\texttt{destination}(e_2) \leftarrow x_{i+1}$
17: $\quad\quad \texttt{edges}(x_i) \leftarrow \texttt{edges}(x_i) \cup \{e_1\} \cup \{e_2\}$
18: $\quad$ **end if**
19: $\quad$ **if** $\texttt{opera}_{op}(\theta)$ '+' **then**
20: $\quad\quad e_1 \leftarrow \texttt{newEdge}(\theta)$, $\texttt{source}(e_1) \leftarrow x_i$, $\texttt{destination}(e_1) \leftarrow x_{i+1}$
21: $\quad\quad e_2 \leftarrow \texttt{newEdge}(\epsilon)$, $\texttt{source}(e_2) \leftarrow x_i$, $\texttt{destination}(e_2) \leftarrow x_i$
22: $\quad\quad \texttt{edges}(x_i) \leftarrow \texttt{edges}(x_i) \cup \{e_1\} \cup \{e_2\}$
23: $\quad$ **end if**
24: $\quad$ **if** $\texttt{eventSelc}_{es}(\theta) =$ ';' **then**
25: $\quad\quad \texttt{sID}_u(\theta) \leftarrow \neg\texttt{sID}_u(\theta)$
26: $\quad\quad e = \texttt{newEdge}(\theta)$, $\texttt{source}(e) \leftarrow x_i$, $\texttt{destination}(e) \leftarrow x_i$
27: $\quad\quad \texttt{edges}(x_i) \leftarrow \texttt{edges}(x_i) \cup \{e\}$
28: $\quad$ **end if**
29: **end for**

(u_2, P_2) be the two GPM expressions then the compilation of SPASEQ temporal operators through the NFA$_{scep}$ automata is described as follows.

- *Immediately Followed-by*: The construction of NFA$_{scep}$ for this operator is the simplest of all, where a single edge for the corresponding state, having different source and destination states, is constructed. The corresponding NFA$_{scep}$ automaton for $((u_1, P_1)$**,**$(u_2, P_2))$ is illustrated in Figure 9.2

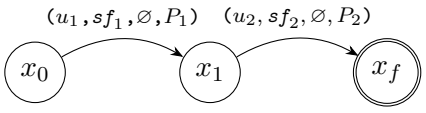

Figure 9.2: Compilation of the *Immediately followed-by* Operator

- *Followed-by*: This operator determines if the irrelevant events has to be skipped or not. Thus, two different edges emanate from the corresponding state, one with the

same source and destination states. This transition matches any kind of event. The corresponding NFA$_{scep}$ automaton for $((u_1, P_1);(u_2, P_2))$ is illustrated in Figure 9.3

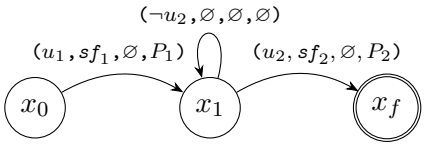

Figure 9.3: Compilation of the *Followed-by* operator

- *Optional*: The optional operator selects an event if it matches to the defined GPM expression, otherwise it ignores the event and moves to the next state. Similarly to the *followed-by* operator, it results in two edges, one with an ϵ-transition. The corresponding NFA$_{scep}$ automaton for $((u_1, P_1),(u_2, P_2)?)$ with the *immediately followed-by* operator is illustrated in Figure 9.4

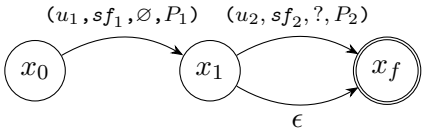

Figure 9.4: Compilation of the Optional Operator

- *Kleene-+*: This operator, as discussed earlier, results in two edges with one edge having the same source and destination state. Thus, it can detect one or more consecutive same events. The corresponding NFA$_{scep}$ for $((u_1, P_1)\ ,\ (u_2, P_2)+)$ with the *immediately followed-by* operator is illustrated in Figure 9.5.

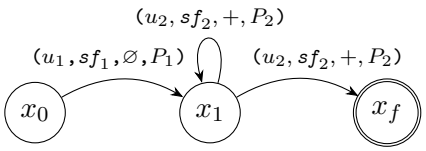

Figure 9.5: Compilation of the Kleene-+ Operator

- *Negation*: This operator detects if either event for a defined pattern does not occur or there is no event occurrence. Thus, it behaves similarly to the optional operators, however, the GPM process is opposite. That is, if an event matches to defined GPM expression, then it violates the condition of the sequence. The corresponding NFA$_{scep}$ automata for $((u_1, P_1),(u_2, P_2)!)$ with immediate followed-by operator (for the sake of brevity) is illustrated in Figure 9.6.

- *Conjunction Operator*: This operator detects the simultaneous occurrence of two or more events. Thus, there are two edges for such state, each destined for a

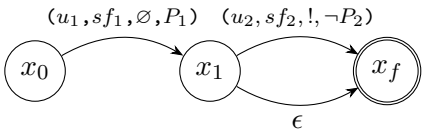

Figure 9.6: Compilation of the Negation Operator

different state. As discussed earlier, we use the concept of manager state for such operator, its functionality is discussed in the next section. The NFA$_{scep}$ automaton for $((u_1, P_1)<>(u_2, P_2))$ is illustrated in Figure 9.7, where x_m is a manager state.

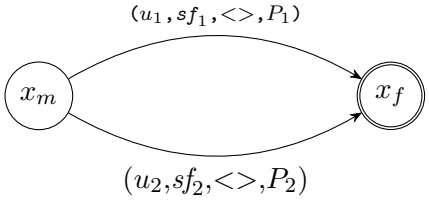

Figure 9.7: Compilation of the Conjunction Operator

- *Disjunction operator*: This operator forms the similar automaton structure as that of conjunction operator, however with the difference of state-transition predicates. The NFA$_{scep}$ automaton for $((u_1, P_1)|(u_2, P_2))$ is illustrated in Figure 9.8

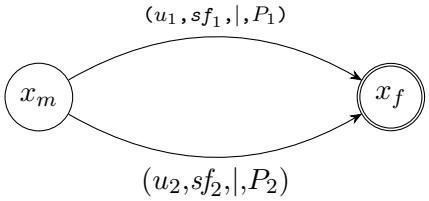

Figure 9.8: Compilation of the Disjunction Operator

It is intuitive enough to see that the compilation process of GPM and sequence expressions corresponds to the semantics defined in Chapter 9, and SPAseq queries are mapped onto equivalent[2] NFA$_{scep}$. Based on these compilation rules, a streamset is used to evaluate the compiled NFA$_{scep}$ automaton. This process is described in the next section.

9.3 System Design of SPAseq Query Engine

Having provided adequate details about the compilation process of SPAseq queries, we finally arrive at the heart of SCEP: the efficient execution of SPAseq queries using NFA$_{scep}$ model. Obviously, this process needs to be optimised, since SPAseq presents additional challenges w.r.t. existing CEP approaches. First, it offers GPM over each RDF graph-based event. Second, it allows static RDF graphs to be joined with the events

[2]Herein, we do not formally define "equivalence" and leave it as our future work. Informally, when a SPAseq query and a NFA$_{scep}$ are equivalent, every portion of the input streamset that produces an output result in the former, will be accepted by the latter and vice versa.

from streams to enrich each of them with more knowledge. This in turn may potentially impact the efficiency of the whole system. Following this intuition, we have implemented various customised optimisations, and reused some from our system called SPECTRA (as discussed in Chapter 7). Herein, we begin with an overview of the underlying components of the SPASEQ query engine, and later provide the main evaluation algorithm for NFA_{scep} automaton. Lastly, we present the optimisation techniques employed by our system.

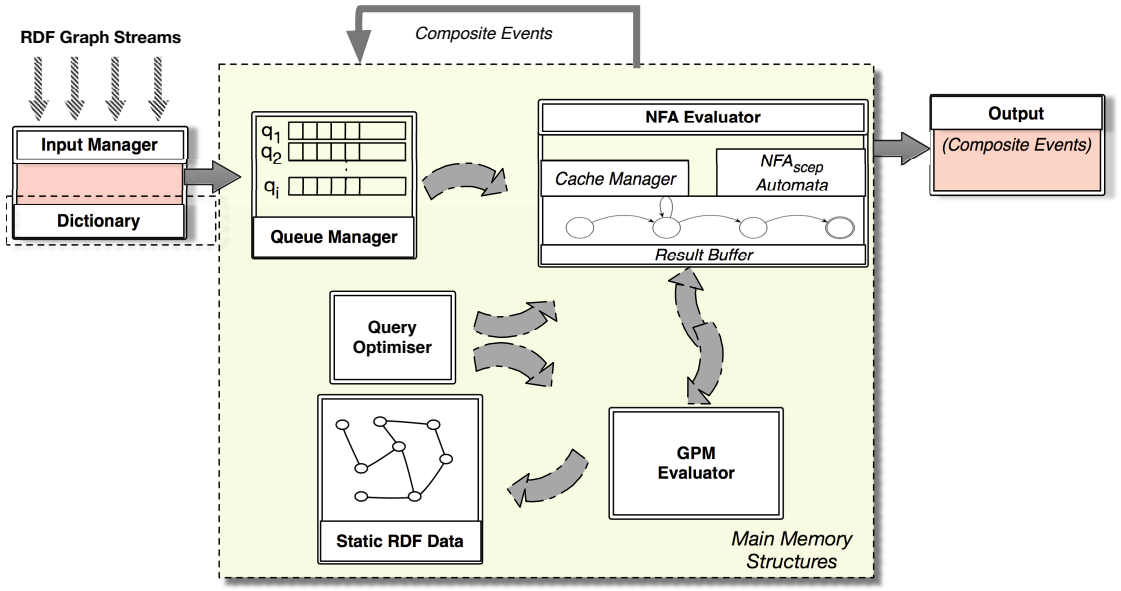

Figure 9.9: Architecture of the SPASEQ Query Engine

Figure 9.9 shows the architecture of the SPASEQ query engine. It resembles with a classical main memory stream processing system. Its main components are *input handler*, *queue manager*, *query optimiser*, *NFA evaluator*, and an RDF engine with *GPM evaluator*. In the following, we briefly discuss these components.

At its heart, the GPM evaluator sticks to the model of SPECTRA engine: it is based on a main memory graph pattern processor but uses specialised data structures and indexing techniques suitable for processing RDF graphs within an event. In particular, it makes use of SUMMARYGRAPH and QUERYPROC operators (see Chapter 7) to first prune the unnecessary triples from each event and then join a set of views (vertically partitioned tables), while utilising incremental indexing. Furthermore, to enrich event data, it can also use the static set of views from the static RDF graphs.

The NFA evaluator contains the compiled NFA_{scep} automaton and employs the GPM evaluator to compute the GPM expressions mapped on the state-transition predicates. Its subcomponent, the *Cache Manager* stores the results of stateful joins, which is also employed by the GPM evaluator. Finally, the *query optimiser* employs various techniques to reduce the load for GPM evaluator and the number of active runs.

The queue manager and input manager do their usual job of feeding the required data into the NFA evaluator. Since our system employs streamset, there are multiple buffers to queue the data from a set of streams. The incoming data from streams are first mapped to numeric IDs using dictionary encoding[3]. The input manager also utilises

[3]Dictionary encoding is a usual process employed by a variety of RDF-based systems [NW10b, Car+04]. It reduces the memory footprints by replacing strings with short numeric IDs, and also increases the

an efficient parser[45] to parse the RDF formatted data into the internal format of the system. The details of the query optimiser are described in Section 9.4

9.3.1 Evaluation of NFA$_{scep}$ Automaton

In this section, we provide an overview of the techniques to evaluate the compiled NFA$_{scep}$ automaton that are implemented by our system. As discussed in Section 9.2.1, the compiled NFA$_{scep}$ automaton represents the model that a matched sequence should follow. Thus, in order to match a set of events emanating from a streamset, a set of runs is initiated at run-time. This set of runs contains partially matched sequences and the run that reaches to its final state represents a matched sequence.

When a new event enters the NFA evaluator, it can results in several actions to be taken by the system. We describe them as follows:

- New runs can be created, or new runs are duplicated from the existing runs in order to cater the non-determinism in NFA$_{scep}$.
- If the newly arrived events match to state-transition predicates of the active states, existing runs can transit from one active state to another.
- Existing runs can be deleted, either because the arrival of a new event results in invalidating the constraints defined in the NFA$_{scep}$ model such as event selection strategies, conjunction, etc., or the selected events in those runs are outside the defined window.

These conditions can be generalised into an algorithm that (i) keeps track of the set of active runs, (ii) starts a new run or deletes the obsolete ones, (iii) chooses the right event for the state-transition predicates, (iv) keep track of stateful joins and their results, and (v) calls the GPM evaluator to match an event with the defined graph pattern.

Algorithm 7 illustrates the execution of an NFA$_{scep}$ automaton. When a compiled automaton is deployed, a single run is initiated from the automaton, which waits at its starting state for the arrival of an appropriate event (*lines* 6–16). With the arrival of a new event, the PROCESSEVENT function does the following: (i) examines (by using the timestamp of the event) if there are runs that are outside the window boundaries, hence to be deleted (*line* 6), (ii) checks if the event is from the same source that the active state of a run is seeking, i.e., compare the stream ids (u) (*lines* 27-28), (iii) employs MATCHPREDICATE to match the current event with the active state's state-transition predicates, i.e., graph pattern P (using GPMMATCH function (*line 24*)), and stores the result in the result buffer, (iv) and finally matches the temporal operators to examine if a state can transit to the next one (*lines 10-15*). The execution of the temporal operators are described as follows:

- *Optional Operator*: The state mapped with optional operators has two outgoing edges, one with the GPM expression, and other with an ϵ-transition. Thus, the algorithm first matches the incoming events with the mapped GPM expression, while computing that the event is from the same required stream. If there is match,

system performance by using numeric comparisons instead of costly string comparisons.

[4]We employed a performance intensive NxParser, which is a non-validating parser for the Nx format, where x = Triples, Quads, or any other number.

[5]NxParser: `https://github.com/nxparser/nxparser`, last accessed: July, 2016.

Algorithm 7 Evaluation of NFA$_{scep}$

```
1: cacheManager ← {}
2: resultBuffer ← {}
3: activeRuns ← {}
4: A ← {x_1, x_2, ..., x_{n+1}}, where next(x_i) = x_{i+1}
5: procedure ProcessEvent(A, cacheManager, resultBuffer, activeRuns, G_e)
6: for each Run r_i ∈ activeRuns do
7:     if checkWindow(r_i, G_e) = true then
8:         x = getActiveState(r_i)
9:         for edges e ∈ edges(x) do
10:            option = MatchPredicate(theta_θ(e), cacheManager, resultBuffer, G_e )
11:            if option = 1 then
12:                currState(r_i) ← next(x)
13:            else if option = 0 then
14:                deleteRun(r_i)
15:            else if option = 2 then
16:                if strategy(theta_θ(e)) = ';' then        ▷ Skip event for Followed-by
17:                    skipEvent(G_e)
18:                else
19:                    deleteRun(r_i)                        ▷ Delete run for Immediately Followed-by
20:                end if
21:            end if
22:        end for
23:    else
24:        deleteRun(r_i)
25:    end if
26: end for
27:
28: procedure MatchPredicate (θ, cacheManager, resultBuffer, G_e)
29: if θ ≠ ϵ && sID_u(θ) = sID_u(G_e) then
30:    if GpmMatch (graphPatt_P(θ), cacheManager, resultBuffer, G_e) then
31:        return 1
32:    else
33:    end if
34: else if sID_u(G_e) = ¬sID_u(θ) then
35:    return 2
36: else
37:    return 0
38: end if
```

the result buffer stores the matched mappings (a hashmap, where the keys are run and state ids, while the values are views or vertically-partitioned tables) and transits to the next state. Otherwise, if the event is not from the desired stream or there does not exist a match between the GPM expression and event, the run takes the ϵ-transition and transits to the next state (*line 11*).

- *Negation Operator*: The negation operator is evaluated in a similar fashion compared with optional operator. However, in this case, the run transits to the next state (producing identity element) if there is no match with the mapped GPM expression. In case, there is a match with the GPM expression, it means run has violated the negation operator and should be deleted (*line 14*).

- *Kleene-+ Operator*: The evaluation of the kleene-+ operator is an interesting one, since the state-transition predicates of the two edges are not mutually exclusive. Thus, to cater the non-determinism for kleene-+ operator a new run is duplicated from the existing one in case of a match. That is, if the newly arrived event is matched with GPM expression of the active state, a new run is cloned from the active run with the same active state (i.e., state with the kleene-+ edges), and the active run transits to the next state. This way the system can keep track of one or more matched events of the same kind (see Figure 9.10).

- *Conjunction Operator*: The case of the conjunction operator is rather complicated: there are two or more outgoing edges – each with a distinct state-transition predicate – and the run should move to the next state if all the state-transition predicates are matched with consecutive events having the same timestamp. Recall from earlier, we use the term *manager state* to compile this operator. Thus, the duty of the manager state is to make sure if all the outgoing edges are computed and the selected events contain the same timestamp. We will see in Section 9.4 how can we optimise such procedure.

- *Disjunction Operator*: The disjunction operator resembles with the optional operator. However, in this case the incoming event has to match with at-least one of the state-transition predicates. Thus, for each active run, the manager state checks the stream id of the event and employs the appropriate edge to match the event with its GPM expression; and if there is a match the run transits to the next state. Note that in this case there could be a number of edges that can match with an event. In Section 9.4, we will see how to optimise the selection of an edge for the disjunction operator.

On top of the above mentioned temporal operators, the evaluation of NFA_{scep} also needs to consider the defined event selection strategies: *followed-by* and *immediately followed-by*. The *immediately followed-by* operator comes natural to the NFA_{scep} model, where the runs are deleted if the subsequent events do not match the defined state-transition predicate. However, the compilation of *followed-by* operator results in an extra edge with the same source and destination state. Therefore, for its evaluation, the system first compare the stream id (u) of the active state's edge and the incoming event. In case the match does not evaluate to true, the system skips the event and stays in the same state (*lines 28-30* in Algorithms 7).

Example 23 *Consider Fig. 9.10. In this example, r_i represents run i, x_0, x_1, x_2, and x_f represent the states using sequence expression* `SEQ (A,B+,C)` *(From* SPASEQ *Query 8.2),*

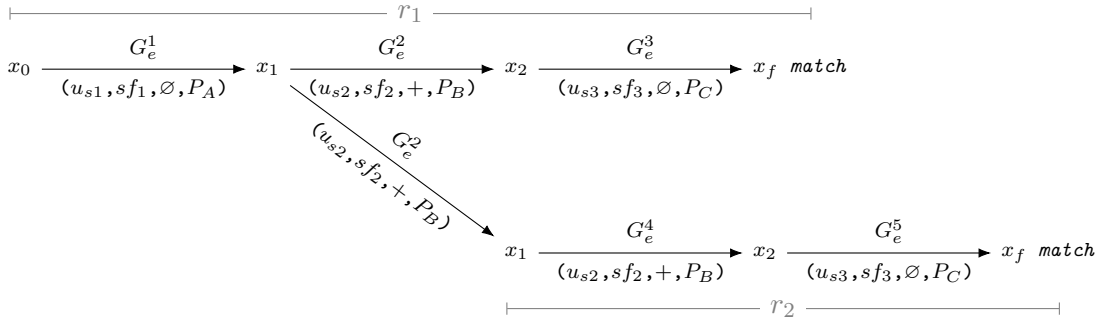

Figure 9.10: Execution of NFA$_{scep}$ runs for the SPASEQ Query 8.2, as described in Example 23

and G_e^k represents an event that occurred at time k. The arrival of G_e^1 results in a new run r_1 and the automaton transits from state x_0 to x_1 (if G_e^1 matches (u_{s1}, P_A)). The next event e_2 results in a non-deterministic move due to kleene-+ operator at the state x_2 (considering G_e^2 matches (u_{s2}, P_B)), and creates a new run r_2 with active state as x_1, while r_1 moves to the next state x_2. When G_e^3 arrives and matches to (u_{s3}, P_C), r_1 moves to the final state and the match is complete. Finally, after the arrival and match of events G_e^4 and G_e^5 with the corresponding GPM expressions, r_2 reaches the final state with a match to the sequence.

9.4 Query Optimisations

The query optimiser is an important component of a CEP system. Generally, the user's query expressed in a non-procedural language describes only the set of constraints a matched pattern should follow. It is up to the query optimiser to generate efficient query plans or adaptively refresh the plans that compute the requested pattern. The two main resources in question for the CEP processing are CPU usage, and system memory: efficient utilisation of CPU and memory resources is critical to provide a scalable CEP system. As discussed in Section 9.1, many different strategies have been proposed to find an optimal way of utilising CPU and memory usage in CEP systems. Thus, one of the main benefits of using an NFA model as an underlying execution framework is that we can take advantage of the rich literature on such techniques. These optimisation techniques can be borrowed into the design of SCEP, while customising them for the RDF graph streams. In this section, we describe how such techniques can be applicable for SCEP, and also proposed new ones considering the query processing over streamset. First, we review the evaluation complexity for the main operators of SPASEQ query language.

9.4.1 Evaluation Complexity of NFA$_{scep}$

The evaluation complexity of NFA$_{scep}$ provides a quantitative measure to establish the cost of various SPASEQ operators. Herein, we first describe the cost of temporal operators in terms of GPM evaluation function, and later provide the upper bound of time-complexity in terms of number of active runs.

Incoming events are matched to the GPM expressions mapped on the state's edges and such evaluation decides if a state can transits to the next one. Depending on the number of distinct GPM expressions mapped to a state, we categorise the operators in two types: *type 1* contains unary operators (negation, kleene-+, optional) and event

selection strategies (*followed-by* and *immediately followed-by*), and *type 2* contains the binary operators (conjunction and disjunction). Let n be the total number of events in a stream (by providing a finite bound over the stream) and k be the number of distinct GPM expressions resulted due to a temporal operator of SPAseq. Then the total cost of GPM evaluation for the type 1 operators is as follows:

$$cost_{type_1} = \sum_{i=1}^{n} \mathbf{c}(P, G_e^i),$$

where $\mathbf{c}(P, G_e)$ represents the cost of matching a graph pattern P with an event G_e^i. The time complexity of such function can be referenced from Theorem 6.1 in Chapter 6.

The cost of type 2 operators can be described as follows:

$$cost_{type_2} = \sum_{i=1}^{n} \sum_{j=1}^{k} \mathbf{c}(P_j, G_e^i),$$

Both of the cost functions can easily be explained through the compilation process of SPAseq operators: type 1 operators results in a single GPM expression to be evaluated, while the type 2 operators, in worst case behaviour, have to match all the GPM expressions for all the edges.

Prior works analysing the complexity of NFA evaluation often consider the number of runs created by an operator and device upper bounds on its expected value [Agr+08, ZDI14]. We adopt the same approach for analysing the complexity of NFA_{scep} evaluation. Thus, for each incoming event, the system has to check all the active runs to determine if the newly arrived event results in (i) state transition from the current active state to the next one, (ii) duplication of a new run, (iii) deletion of the active run if the event violates the defined constraints. Therefore, query operators that result in creating new runs or those who increase the number of active runs are considered to be the most expensive ones. In order to simplify the analysis, we make the following assumptions:

1. We ignore the cost of evaluating GPM expression over each event.
2. We ignore the selectivity measures of the state-transition predicates, i.e., the events that are not matched and either skipped or result in deleting a run of an NFA_{scep} automaton. Hence, focusing on the worst-case behaviour.

Based on this, let us consider that n events arrive at a current active state of a run, where the active state may contain the following set of operators: *followed-by*, *immediately followed-by*, negation, kleene-+, optional, conjunction and disjunction.

Theorem 9.1

The upper bound of evaluation complexity of immediate followed-by, followed-by*, negation, and optional is linear-time $\mathcal{O}(n)$, where n is the total number of runs generated for the n input events.*

Proof Sketch. It is intuitive enough to observe that none of the operators discussed above duplicate runs from the existing ones, and each has only one GPM expression to be matched with the incoming events. Let us consider the case of event selection operators. Given a sequence expression $((u_i, P_i)\ op\ (u_j, P_j))$, where $op = \{\ ',', ';'\}$, mapped to states x_i

and x_j. With the arrival of an event G_e at τ, where an event selection operator is mapped at state x_j, it can result in the following actions: (i) the run will transit to the next state, (ii) the event will be skipped due to *followed-by* operator, and (iii) the run will be deleted. Since we are considering the worst-case behaviour, let us dismiss the situations (ii) (iii). In situation (i) there be will no extra run created for the above mentioned operators, and each incoming event will be matched to only one GPM expression. Thus the evaluation cost remains linear. The same is the case with the other operators (negation, optional). An event that matches to these operators will never result in duplication of a run. If these operators are mapped at the first state (i.e., negation and optional) each matched event from the stream will create a new run, thus for n number of events there can be only n runs.

Although, the upper bound of above mentioned operators has the same evaluation complexity, there exists discrepancies when considering the real-world scenarios.

Immediate followed-by Vs Followed-by: As discussed previously, the *followed-by* operator skips irrelevant events, while *immediate followed-by* is highly selective on the temporal order of the events. Thus, the duration of a run is largely determined by the event selection strategy. Due to the skipping nature of *followed-by* operators, the life-span of its runs can be longer on a streamset. In particular, those runs that do not produce matches, and instead loop around a state by ignoring incoming events until the defined window expires. On the contrary, the average duration of a run is shorter for the *immediately followed-by* operator, since a runs fails immediately when it reads an event that violates its requirements. Such difference in their evaluation cost is visible in our experimental analysis (Section 9.5).

The case of conjunction and disjunction operators is slightly different, and therefore has a different upper case bound. That is, for each conjunction/disjunction operator, there are more than one edges with the following properties: (i) more than one edge have different source and destination states with distinct GPM expressions, (ii) the edges do not have an ϵ-transition. Thus, in worst case each incoming event has to match to the complete set of state-transition predicates. Hence, for k such edges for conjunction/disjunction operator, and n input events, we can provide the upper bound on the complexity of these operators as follows:

Theorem 9.2

The upper bound of evaluation complexity of conjunction and disjunction is $\mathcal{O}(n \cdot k)$, where n is the total number of runs generated and k is number of GPM expressions mapped to the manager state.

Proof Sketch. Comparing the number of runs generated by the conjunction and disjunction operators, we can infer from Theorem 9.1 that the total number of runs generated is bounded by n. That is, if an event arrives at the active state x_m (a manager state for conjunction/disjunction), it either matches to the set of edges defined and moves to the next state, or it stays at the current active states and waits for new events (in case of conjunction). Therefore, no new runs are generated for these operators. However, for both of these operators, the manager state has to choose from a set of edges and compare each incoming event with the compatible edges. That is, the edges whose state-transition predicates are waiting for the event with a stream id (u_i). Thus, even if the number of

active runs remains the same, each event may have to be matched with a number of GPM expressions. For k edges, in worst case each event has to match the k edges.

Theorem 9.3

The upper bound of evaluation complexity of kleene-+ operator is quadratic-time $\mathcal{O}(n^2)$, where n is the number of runs generated by n events.

Proof Sketch: Consider a sequence expression $((u_j, P_j)+)$, which is mapped onto the state x_j with the kleene-+ operator . Let us consider that x_j is an active state. If an event G_e arrives at time τ, and if it matches the GPM expression of the state-transition predicate, it will duplicate the current active run and append the duplicated one to the list of active runs. Thus, for each newly matched event at a kleene-+ state, a new run is added to the active list, and for n such events, there will be in total n^2 runs to be generated considering all the event are matched to the GPM expression of the state x_j, i.e., worst case behaviour.

The kleene-+ operator is the most expensive in the lot, in terms of number of active runs. Hence, based on the observations in Theorem 9.1 and 9.3, we adopt some of the optimisation strategies previously proposed, and also propose some new ones. We divide these techniques into two classes from the view point of operators and system: *local*, and *global* levels. Local-level optimisation techniques are targeted at the specific operators considering their attributes, while the global-level optimisation are for all the operators, and are implemented at the system level. In the following section, we present these techniques in details.

9.4.2 Global Query Optimisations

The evaluation of an NFA$_{scep}$ automaton is driven by the state-transition predicates being satisfied (or not) for an incoming event. The number of active runs of an NFA$_{scep}$ automaton, and the number of state-transition predicates that each run could potentially traverse can be very large. Hence, traversing these runs for each incoming event is not feasible. Therefore, the aim of global optimisation is to reduce the total number of active runs by (i) deleting, as soon as possible, the runs that cannot produce matches, and (ii) indexing the runs to collect the smaller number of runs that can be affected by an event.

Pushing the Temporal Window

As mentioned, the window defined in a SPAseq query constraints the matches to be executed over unbounded streams. The implication of this is that it is desirable to evict the runs that contain older events as soon as possible. The timestamp of the newly arrived event is used to update the boundaries of the defined window, and to delete the runs that are outside the window. Therefore, before processing each event, if we push the evaluation of the temporal window at the top of the processing stack, we can delete the runs without first evaluating the state-transition predicates, consequently decreasing the size of active runs's list. In Algorithm 7 (*line 7*), we push the window check before iterating over the active run list. This allows older runs to be evicted as soon as they fail to satisfy the window constraint [ZDI14].

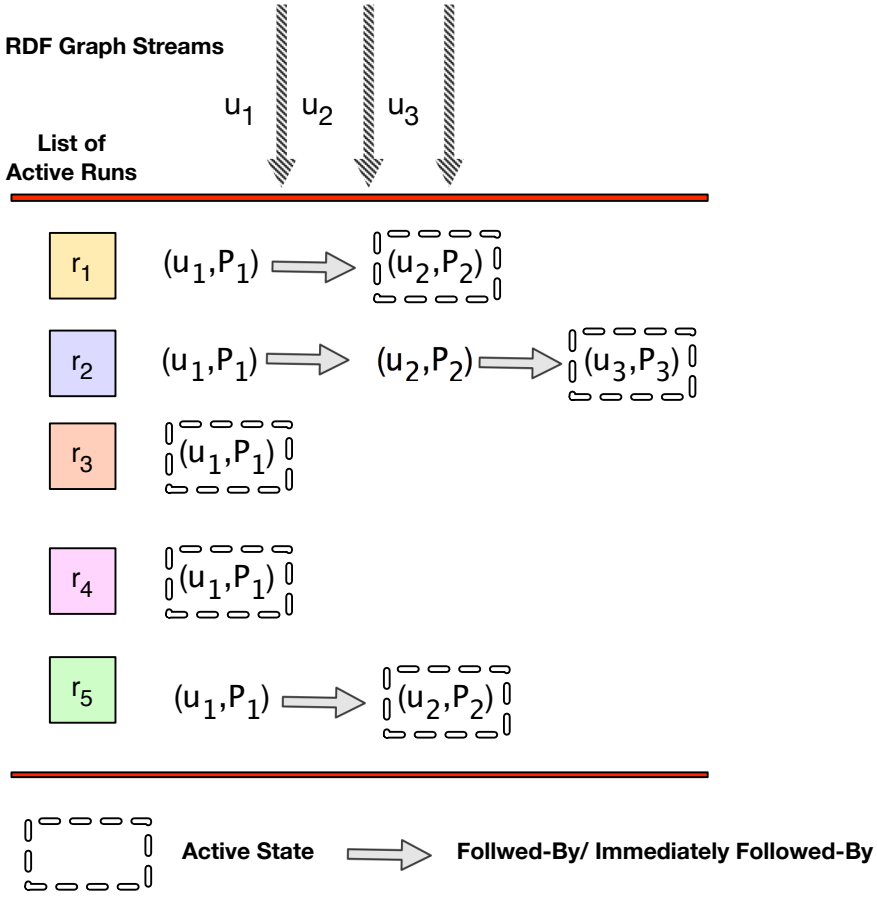

Figure 9.11: Processing Streamset over Active Runs

Pushing the Stateful Predicates

The stateful predicates define the joins between a set of graph patterns. These joins can be defined through the `FILTER` expression or within the set of graph patterns (see SPASEQ Query 8.2). With the arrival of an event, there are two obvious possible ways to invoke the GPM evaluator. First, we can evaluate the event with the defined GPM expression, and later use the cache manager to perform the stateful joins. However, this would require us to issue an expensive GPM process, and if the stateful joins are not fruitful, GPM would be of waste and we have to either delete the run or skip the event (depending upon the defined operator). Alternatively, we can first use the cache manager to implement the stateful joins, and only then employ the complete GPM against the event. This would allow us to prune the irrelevant events without initiating the complete GPM process. Moreover, this would also result in decreasing the intermediate result set, which consequently reduces the load over the GPM evaluator. Our system employs the second approach and pushes the stateful joins as early as possible in the processing stack.

As an example of this, consider the SPASEQ Query 8.2. Its GPM expressions share the stateful variables of location (`?l`) and electricity fare (`?fr`). Pushing these two joins, we can easily ignore the events that would not contain the expected mappings of these variables, and consequently the system does not have to process the complete GPM expressions (`GPM B` and `C`) for such events.

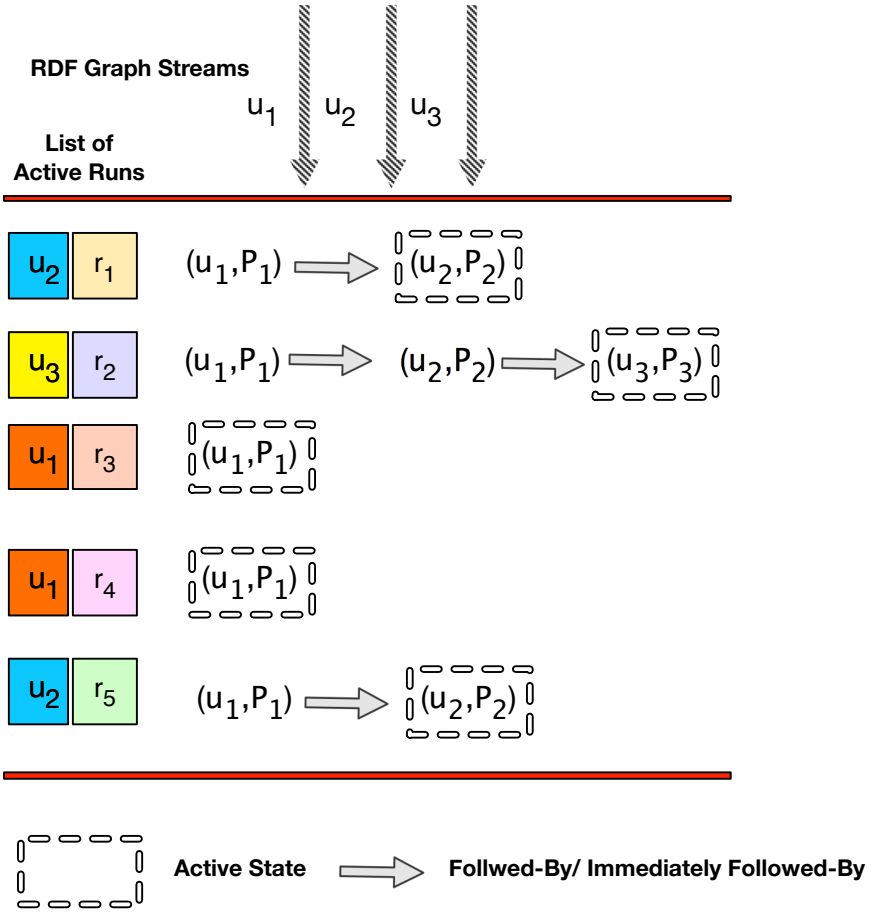

Figure 9.12: Partitioning Runs by Stream Ids

Indexing Runs by Stream Ids

SPAseq queries are evaluated over a streamset, which means that the edges from each state contain the stream id that is used to match the graph patterns. Therefore, each active state waits for a specific type of event from a specific stream, and later invokes the GPM evaluator. We illustrate it through an example.

Example 24 *Figure 9.11 shows a set of streams employed by a set of active runs. The active runs r_1, r_5 are waiting for an event from stream with id u_2, the active run r_2 is waiting for an event from stream u_3, and active runs r_3 and r_4 are waiting for the events from the stream with id u_1. According to Algorithm 7, for each incoming event from the set of streams, we have to iterate over all the active runs and then match the stream id and GPM expressions respectively.*

The goal of indexing runs by the stream ids is to efficiently identify the subset of runs that can be affected by an incoming event: although, the total number of active runs can be very large at a given time, the number of runs affected by an event is typically lower. Thus, we index each run by the stream id of its active state (see Figure 9.12). More precisely, the index takes the stream id as a key and the corresponding run as the value. These indexes are simple hash tables, and for each incoming event it essentially returns a set of runs that can be affected by the incoming event. Nevertheless, these indices

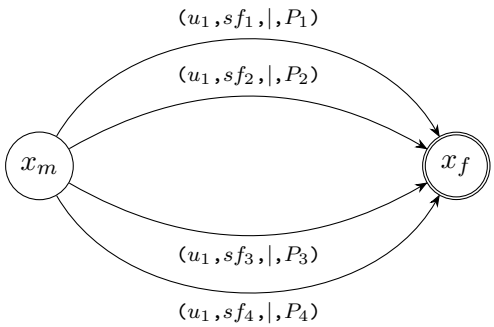

Figure 9.13: Compilation of Disjunction Operator for $((u_1, P_1) \mid (u_1, P_2) \mid (u_1, P_3) \mid (u_1, P_4))$

proved to be a useful feature for processing events from a streamset. Note that the naive implementation using a single list of runs would be inefficient: each incoming event would iterate over all the active runs, and initiate the matching process for each of them.

Memory Management

Although in-memory data access is extremely fast compared to the disk access, an efficient memory management is still required for a SCEP: the data structures usually grow in proportion to the input stream size or the matched results. Thus, events that are outside the defined window, or that cannot produce a match must be discarded in order avoid the unnecessary memory utilisation. The three main data structures that require tweaking are *cache manager*, *result buffer* and *active run list* or set of indexed runs. In this context, our first step is to use the *buy bulk* design principle. That is, we allocate memory at once or infrequently for resizing. This complies to the fact that the dynamic memory allocation is typically faster when allocation is performed in bulk, instead of multiple small requests of the same total size. Second, since the cache manager and the result buffer are indexed with the dynamically generated run ids, we use the expired runs – which either is complete or not – to locate the exact objects to be deleted. These objects are added to a pool: when a new object is created, we try to recycle the memory from such pool. This limits the initialisation of new objects and reduces the load over the garbage collector. Note that we use hash-based indexing for all the data structures, which means the position of expired objects can be found in theoretically constant time.

9.4.3 Local Query Optimisation

Local query optimisation is devised for the conjunction and disjunction operators, where the chief problem is how to select the GPM expression from a set of edges and how to reduce the load on the GPM evaluator. Thus, the knowledge of the runs affected by an incoming event is not sufficient, we also have to determine which edge these runs will traverse.

To better illustrate the problem, let us start by examining the sequence expression $((u_1, P_1) \mid (u_1, P_2) \mid (u_1, P_3) \mid (u_1, P_4))$ for the disjunction operator. Figure 9.13 shows the related NFA_{scep} automaton. Now consider an input stream u_1, and an event G_e^i at time τ_i for the manager state x_m. In order to process such event, the manager state has to choose from a set of state-transition predicates and direct the event and selected graph pattern P to the GPM evaluator: the manager state cannot be picky, as all the mapped predicates require the event from stream u_1. Now the question is how to choose the less costly graph pattern to be selected by the manager state.

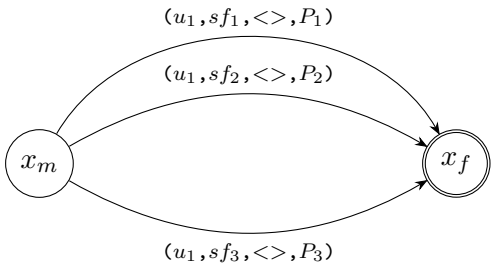

Figure 9.14: Compilation of Conjunction Operator for $((u_1, P_1) <> (u_1, P_3) <> (u_1, P_3))$

The optimal way of processing the optional operator would be to sort the graph patterns according to their cost, and select the cheapest one for the first round of evaluation. That is, if $\mathbf{c}(P_i, G_e^i)$ is the cost of matching a graph pattern with an event G_e^i, then we require a sorted list such that $\mathbf{c}(P_j, G_e^i) < \mathbf{c}(P_k, G_e^i) < \cdots < \mathbf{c}(P_m, G_e^i)$. The question is how to determine the cost of GPM evaluation. There can be two different ways to it.

1. Use the selective measures and structure of the graph patterns. That is, how much the GRAPHSUMMARY operator (from SPECTRA Chapter 7) be handy for them.

2. Adaptively gather the statistics about the cost of matching a specific graph pattern, and sort the graph pattern accordingly.

Let us focus on the first approach. As discussed in Chapter 6, the cost of matching a graph pattern and an event is directly proportional to each of their sizes. That is, if there are more triple patterns $tp \in P$, then there will be more join operations on different vertically-partitioned views: this can give us fair bit of idea about the costly graph patterns. Furthermore, due to the presence of filters, the GRAPHSUMMARY operator can prune most of unnecessary triples, and consequently reduces the cost of the GPM operation. Following this reasoning, we keep a sorted set of graph patterns $\{P_1, P_2, \ldots, P_k\}$ within the manager state, each associated with a stream id u_i. This set is sorted by checking the number of triple patterns, and the selectivity of subjects, predicates and objects within a graph pattern during the query compilation (as discussed in Chapter 7). The manager state can utilise this set to first inspect the less costly graph patterns for the incoming events. This can lead to a less costly optional operator with few calls to the GPM evaluator.

The second approach is based on the statistics measures. That is, the system during its life-span observes which graph pattern has been utilised successfully in the past and is less costly compared with others. This approach can be built on top of the technique discussed above. Herein, the implementation of such optimisation technique is considered as future work.

The conjunction operator, however, contains an additional challenge on top of the one discussed above. To illustrate this, let us consider a sequence expression $((u_1, P_1) <> (u_1, P_2) <> (u_1, P_3))$ for the conjunction operator. Figure 9.14 shows the NFA_{scep} automaton for it. In order for the manager state x_m to proceed to the next state, it has to successfully match all the defined state-transition predicates, such that events satisfying them should occur at the same time. Thus, if an event G_e^i arrives and matches to one of the state-transition predicate, the automaton buffers its result, timestamp, and waits for the remaining events. Now consider a situation, where events G_e^i and G_e^j arrive at τ_1 and match with the GPM expressions (u_1, P_1) and (u_2, P_2) respectively. Then the automaton waits for an event to satisfy GPM expression (u_3, P_3). Now consider an

event G_e^m arrives at τ_2. It results into two constraints to be examined (i) if $\tau_1 = \tau_2$, and (ii) if G_e^m matches with the GPM expression (u_3, P_3). Here, if any of the above mentioned constraints would not match, then it means the run has to be deleted and all the previous GPM evaluations were useless: the process of matching an event with a graph pattern is expensive and it stresses the CPU utilisation.

Our approach to address this issue is to employ a *lazy evaluation* technique. Conceptually, it delays the evaluation of graph patterns until it gets enough evidence that these matches would not be useless. Its steps are described as follow:

1. Buffer the events from streams until the number of events with the same timestamps is equal to the number of edges (with distinct GPM expressions) going out from the manager state.

2. After the conformity of the above constraint, choose graph patterns according to their costs (as discussed for disjunction operator).

Algorithm 8 Evaluation of NFA$_{scep}$

```
1: INPUT: cacheManager, resultBuffer
2: eventBuffer ← {G_e^1, G_e^2, ..., G_e^i}, where τ_1 = τ_2 = ··· = τ_i
3: procedure ProcConjunction(x_m, eventBuffer, cacheManager, resultBuffer)
4: if size(edges(x_m)) = size(eventBuffer) then
5:     sortedEdges ← getSortedEdges(edges(x_m))
6:     for edges e ∈ sortedEdges do
7:         stop ← true
8:         for event G_e ∈ eventBuffer do
9:             if GpmMatch(graphPatt_P(Theta_θ(e)), cacheManager, resultBuffer, G_e)
   = true then
10:                stop ← false
11:                removeEvent (eventBuffer, G_e)
12:                break the loop
13:            end if
14:        end for
15:        if stop = true then
16:            break the loop
17:        end if
18:    end for
19:    if size(eventBuffer) = ∅ then
20:        return true
21:    else
22:        return false
23:    end if
24: end if
```

The main idea underlying our lazy evaluation strategy is to avoid unnecessary high cost of GPM, and to start the GPM process when it is probable enough that it would return the desired results. The idea of lazy batch-based processing is the reminiscent of earlier work on buffering the events and process them as batches [MM09].

Algorithm 8 shows the lazy evaluation of the conjunction operator. The ProcConjunction function takes the cache manager, result buffer and an event buffer that

contains a set of events with the same timestamps. It first examines the size of both buffered events and the number of edges mapped to a manager state x_m (*line 4*). If this check evaluates to true, it extracts the set of sorted edges, while considering the selectivity of graph patterns (*line 5*). The algorithm then iterates over the set of edges and matches them with the buffered events. If an event $G_e \in$ *eventBuffer* matches the graph pattern P, the algorithm removes this event from the event buffer (*lines 8-13*). The whole process continuous and if all the events ($G_e \in$ *eventBuffer*) are matched with the defined edges of the manager state, the automaton transits to the next state (*lines 19-20*). Otherwise, it either deletes the run for the *immediately followed-by* operator, or it waits for new events to be filled in the buffer for the *followed-by* operator.

In this section, we presented various optimisation strategies employed by SPAseq. The focal point of the global optimisations is to reduce the load for the GPM evaluator (1) by pushing the temporal window such that the events that are outside the window boundaries are not sent to the GPM evaluator, (2) by pushing the stateful predicates to prune the events that are unlikely to match, and to reduce the size of the intermediate results for GPM. Furthermore, we also index runs by the stream ids such that only the runs that can be affected by the incoming events are selected. The goal of the local optimisations is the efficient evaluation of the conjunction/disjunction operator. That is, using the GPM evaluator for a batch of likely events that can result in a match. In the proceeding section, we present the experimental evaluation of such optimisation strategies.

9.5 Experimental Evaluation

In this section, we present the experimental evaluation that examines (i) the complexity of various SPAseq temporal operators, (ii) the effect of various optimisation strategies, and (iii) the comparative analysis against state-of-the-art systems. We first describe our experimental set up, and later we analyse the system performance in the form of questions. Our system, called SPAseq, is implemented in Java, and to support reproducibility of experiments, it is released under an open source license[6].

9.5.1 Experimental Setup

Datasets: We used one synthetic and one real-world dataset, and their associated queries for our experimental evaluation.

The Stock Market Dataset (SMD) contains share trades information in the stock market. In order to simulate the real-world workload and properties of stock prices, we use the random fractal terrain generation algorithm [KKM13, Sim+16]: it is based on the fractal time series and provides properties such as randomness, non-determinism, chaos theory, etc. Our SMD data generator is openly available at our project website. We generated a dataset of more than 20 million triples and 10 million RDF graph events.

The UMass Smart Home Dataset (SHD) [WLC14] is a real-world dataset and provides power measurements that include heterogeneous sensory information, i.e., power-related events for power sources, weather-related events from sensors (i.e., thermostat) and events

[6]SPAseq: `http://spaseq.github.io/`, last accessed: July, 2016.

for renewable energy sources. We use a smart grid ontology [GLP+14] to map the raw eventual data into N-Triples format for three different streams: the *power stream* ($\mathcal{S}_{g_1}$), the *power storage* stream ($\mathcal{S}_{g_2}$) and the *weather stream* ($\mathcal{S}_{g_3}$). In total the dataset contains around 30 million triples, 8 million events.

Queries: We use two main queries for the above mentioned datasets: **UC** 2 (Query 8.2) and **UC** 1 (Query 8.3). That is, a V-shaped pattern and a smart grid pattern. Both of these queries are further extended for various experiments.

Constraints: The execution time/throughput of the systems includes the time needed to load and parse the streams. It also includes the time needed to parse the output into a uniform format, and time for writing results to disk. For each experiment, the maximum execution time is limited to two hours and the maximum memory consumption to 20 GB. The former is enforced by external termination and the latter by the size of the JVM. For robustness, we performed 10 independent runs of each experiment and we report the median values.

Stream Configurations: We use two different configurations to generate streams for both datasets.

- *Config. 1*: Random generation of events, i.e., events are generated according to the real-world conditions.
- *Config. 2*: Sequence-based generation of events, i.e., events are generated according to the sequence defined in the SPAseq queries. This results in maximum number of matches to be produced and allows to determine the worst-case behaviour of various temporal operators.

Hardware: All the experiments were performed on Intel Xeon E3 1246v3 processor with 8MB of L3 cache. The system is equipped with 32GB of main memory and a 256Go PCI Express SSD. The system runs a 64-bit Linux 3.13.0 kernel with Oracle's JDK 8u05.

9.5.2 Results and Analysis

We start our analysis by first describing the evaluation cost of SPAseq operators, and how they effects the performance of the system, later we present the usefulness of various optimisation strategies, and finally we provide the comparative analysis of SPAseq and EP-SPARQL for the same dataset and use cases.

Comparative Analysis of SPAseq Operators

Question 1. *How does the unary operators kleene-+, negation and optional perform w.r.t to each other?*

First we describe the set up for this set of experiments. In order to compare the relative complexity of the operators, we employed SMD dataset, **UC 2** queries, and *Config. 2* to generate streams. It allows to make sure that each operator in question has the exact number of matches. We used the *immediately followed-by* operator between unary operators, since the performance differences between the *followed-by* and *immediately followed-by* operators is mainly visible in a multi-stream environment.

Figure 9.15 shows the results of our experiments. As expected, the negation and optional operators show linear scaling behaviour with the increase in the window size.

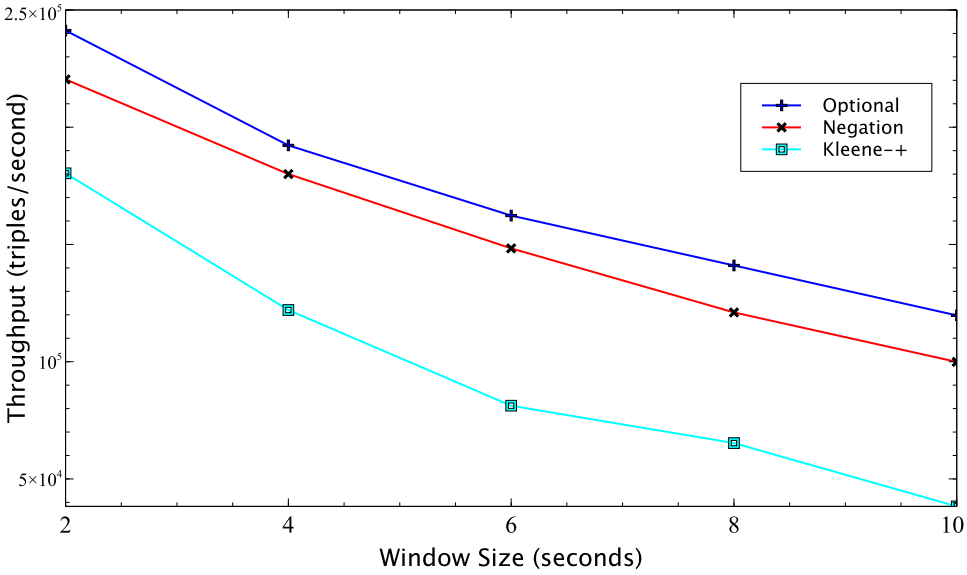

Figure 9.15: Performance Measures of Optional, Negation and Kleene-+ Operators

That is, the number of runs generated for each of them is relatively proportional to the number of events matched at the first state's state-transition predicate. However, the same is not the case with the kleene-+ operator. If an event matches to the kleene-+ operator, the system duplicates an additional run and adds it to the active runs list. This means, following the same intuition from earlier, each newly arrived event has to process a large number of active runs. This results in extra cost for kleene-+ operator to process an event.

Another important point, that can be inferred from Figure 9.15, is that the negation operator is slightly more expensive than the optional operator. This is because if an event does not match to the negation state's state-transition predicate, it is again used to check if it can match with the next state's state-transition predicate (if it is not the final state). Thus resulting in an extra evaluation of a GPM expression. Nevertheless, the aforementioned set of experiments showcases the result parallel to the one discussed for the evaluation complexity of SPASEQ operators (Section 9.4.1).

Question 2. *How do the binary operators conjunction and disjunction perform w.r.t to each other?*

For this set of experiments, we again used SMD dataset, **UC 2** queries, and *Config. 2* to generate streams. As all the events emanates from a single stream, it provide the most complex case: since all the GPM expressions mapped at the conjunction/disjunction state's edges expect the events with the same stream id, the system cannot be choosy and, in worst case, all the events arrived at conjunction/disjunction state have to be matched with all the mapped GPM expressions. Figure 9.16 shows the evaluation of these operators while increasing the number of edges (k) for the conjunction/disjunction states, and having a fixed window size of 5 seconds. Since, as observed in the complexity analysis of these operators, their performance degrades linearly with the increase in the number of edges, such a behaviour can be confirmed from Figure 9.16. As expected, conjunction operator scale linearly with the increase in number of edges to be matched, while the disjunction operator scale sub-linearly, since it has to match only one of the edge to transit to the next state. Moreover, not surprisingly, conjunction operator is much more expensive than the disjunction operator. The conjunction operator has to match the

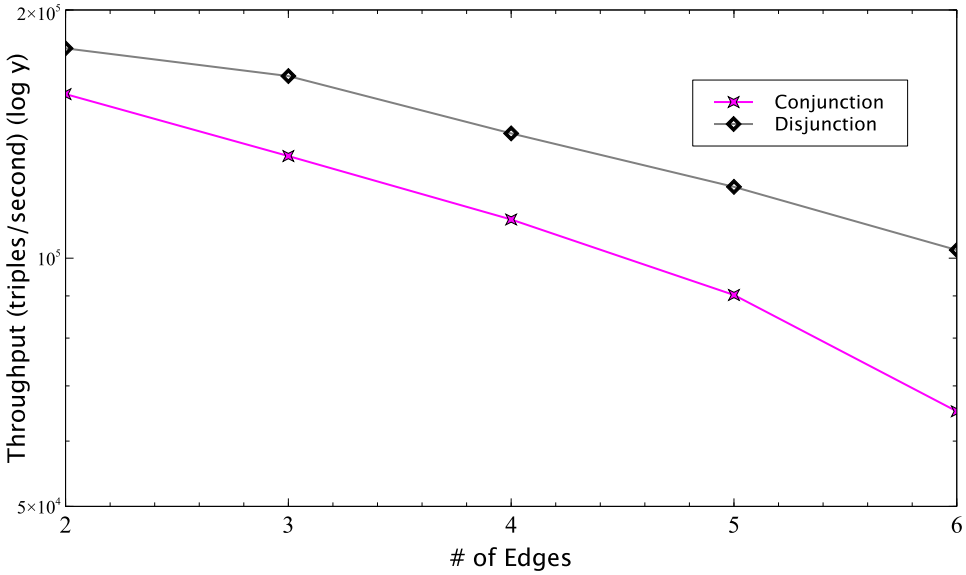

Figure 9.16: Comparison of *Conjunction* and *Disjunction* Operators

complete set of GPM expressions mapped on the set of edges, while disjunction results in few matches and its corresponding state transits to the next one if there is a match with either of them. Note that, for this set of experiments, we use the lazy evaluation strategy for conjunction operator, it comparison with the eager strategy is provided in *Question 5*.

Question 3. *How do the event selection strategies immediately followed-by and followed-by perform w.r.t to each other?*

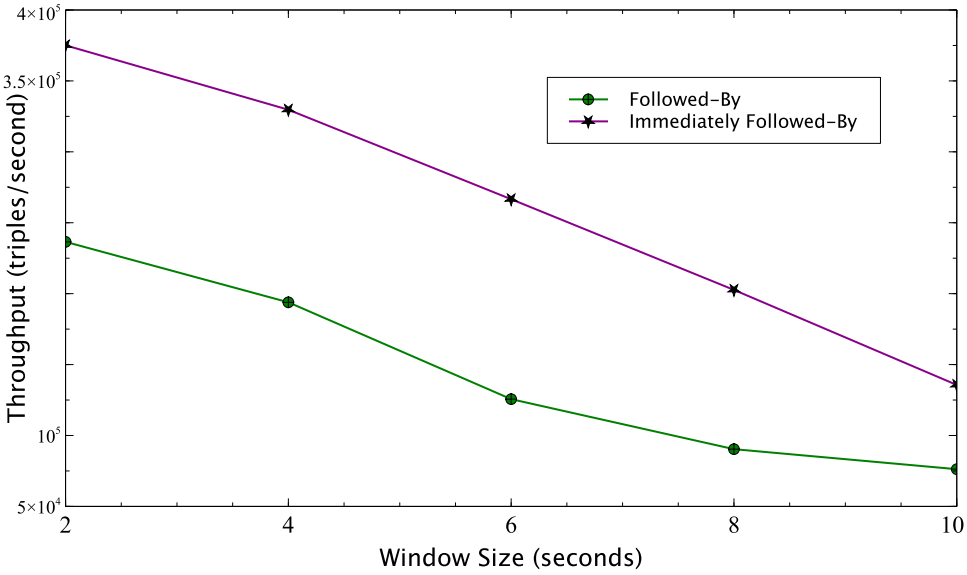

Figure 9.17: Comparison of *Followed-by* and *Immediately Followed-By* Operators

For this set of experiments, we use the SHD dataset and **UC** 1 queries, since the effect of the operators in question is distinguishable in a multi-stream environment. We tested both operators using only *Config. 1*, since *Config. 2* produces events according to the defined patterns, hence no events are skipped for the *followed-by* operator and both

operators will have the similar performance measures. Figure 9.17 shows the evaluation of these operators using *Config. 1*. As we can see that the *immediately followed-by* operator is less expensive compared with the *followed-by* operator: *followed-by* operator skips the irrelevant events (due to the random generation of events) that are not destined for the active state of a run, while the *immediately followed-by* operator delete the run as soon as it violates its defined constraint. The runs for the *followed-by* operator are only deleted if (i) the timestamps of the selected events for the run is outside the defined window, or (ii) there is no match with an event that emanates from the same stream defined for the current active state's edge. Thus, the average life-span of runs for *followed-by* operator is far more than the *immediately followed-by*. This means, for each event the system has to go through a considerably large list of runs to be matched.

Effects of Optimisation Strategies

Question 4. *How does the indexing runs by stream ids strategy affects the performance of the system?*

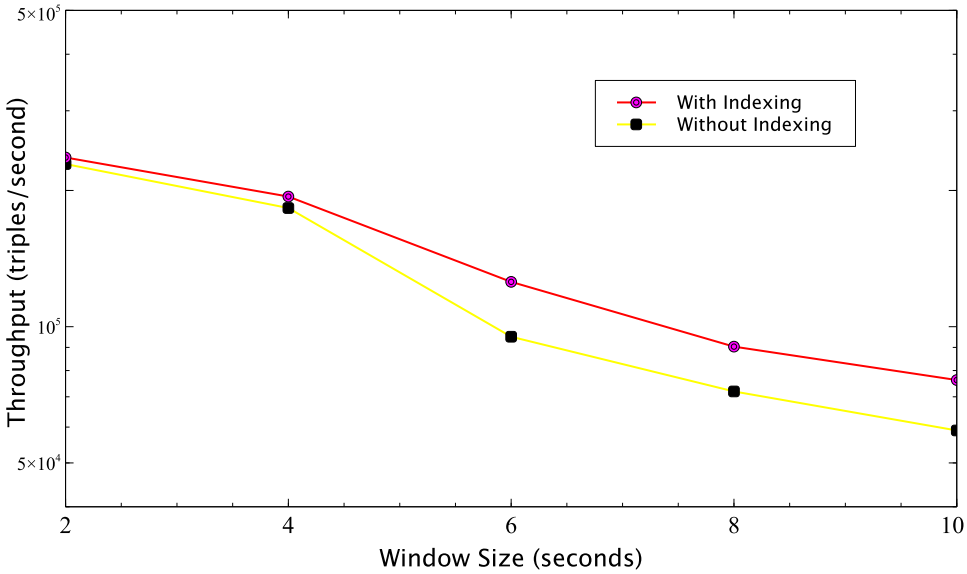

Figure 9.18: Analysis of Indexing Runs by Stream Ids

In order to determine the effectiveness of indexing runs by stream ids, we employ the SHD and **UC 1** query with *followed-by* operator, and *Config. 1*: since the *followed-by* operator is costly compared with the *immediately followed-by* operator. Recall from Section 9.4.2, we index runs by stream id, thus when an event arrives, only the runs whose active state is waiting for such an event is used from the complete list of active runs. Consequently, it reduces the overhead of going through the whole list of available active runs. Figure 9.18 shows the results of our evaluation with variable window sizes. According to the results, the performance differences between the indexed and non-indexed approach is not evident at smaller windows. This is due to the fact that small number of runs are produced/remains active for the smaller windows, hence indexing of runs does not results in a comparatively smaller set of runs to be probed for each event. However, the effectiveness of the indexing technique becomes quite clear with the increase in the window size. That is, a large number of runs are produced with a smaller set of them waiting for an event for a specific stream. For instance, if an event G_e^i arrives at time

τ_i from a stream with id u, then only the runs whose active states are waiting for the event from stream u – hashed indexed by stream id u – are collected, and input event is probed against the state-transition predicates of their active states.

Question 5. *How does the lazy evaluation affects the performance of the conjunction operator?*

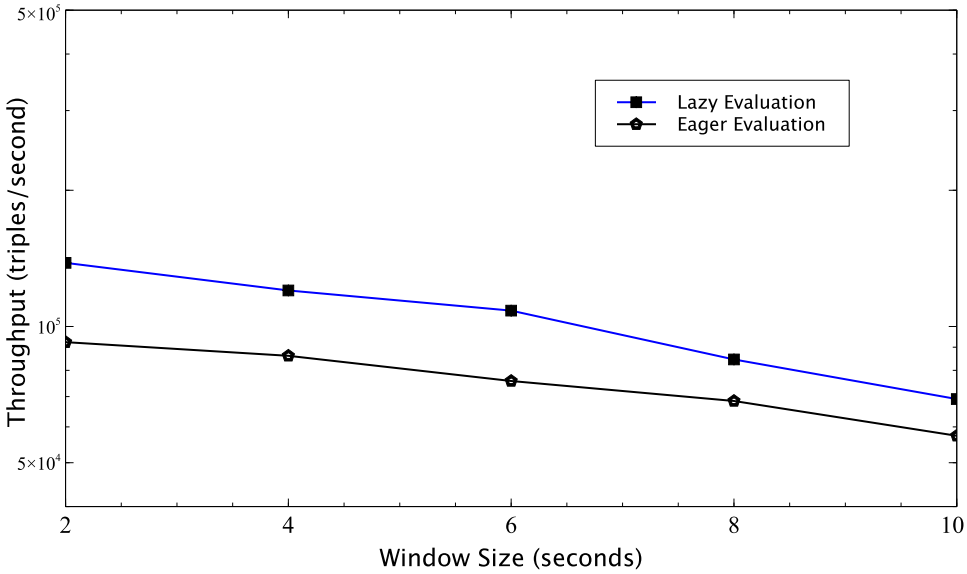

Figure 9.19: Lazy vs Eager Evaluation of Conjunction Operator

For this set of experiments, we again employ SMD dataset, **UC 2** (with conjunction operator containing 4 edges), and *Config. 1* to generates events: *Config. 2* produces events according to the defined pattern and the effects of the lazy evaluation would not be obvious in such case. Figure 9.19 shows the results of the conjunction operator with lazy and eager evaluation strategy. Recall from Section 9.4.3 lazy evaluation delays the computation of all the state-transition predicates until the number of the events with the same timestamp is equal to the number of state-transition predicates. As shown in Figure 9.19, lazy evaluation performs much better on smaller windows and relatively better on larger ones: eager evaluation results in a larger number of useless calls to the GPM evaluator, while lazy evaluation performs a batch-based calls to the GPM evaluator. Thus, with lazy evaluation, a set of events are evaluated against a set of GPM expressions, only if all the buffered events (within a manager state) has the same timestamp. For the smaller window, if there are not enough of such events, there are no calls to the GPM evaluator, and with the expiration of the window, the run is deleted. Contrary to this, eager evaluation call the GPM evaluator for each incoming event and a larger number of such calls proved to be useless for smaller windows.

Comparative Analysis with EP-SPARQL

Question 6. *How does the* SPASEQ *engine perform w.r.t the EP-SPARQL engine?*

Before describing the results, we first presents some of the assumptions for our comparative analysis. Both SPASEQ and EP-SPARQL differ w.r.t each other in terms of semantics and data model. Hence, they may produce different results for the same query. Therefore, the aim of our comparative analysis is to employ the same use case,

its queries and dataset to measure the performance differences between the two. This strategy is mostly utilised by the information retrieval systems.

For the dataset and queries, we used SMD dataset, its respective query for the V-shaped pattern and *Config. 2* to produce the maximum number of matches. For the first set of experiments, we used a simple V-shaped pattern query while increasing the window size, and later we used the same pattern while varying the number of sequence clauses or elements in the sequence expression with fixed window size of 8 seconds. Note that, since we used *Config. 2* for this set of experiments (sequence-based event generation), both *followed-by* and *immediately followed-by* operators will have same performance measures. Furthermore, we used a simple V-shaped pattern query, since EP-SPARQL does not support kleene-+ operator (see Chapter 8 (Section 8.7.3)). Sample queries are presented in Appendix A.5.

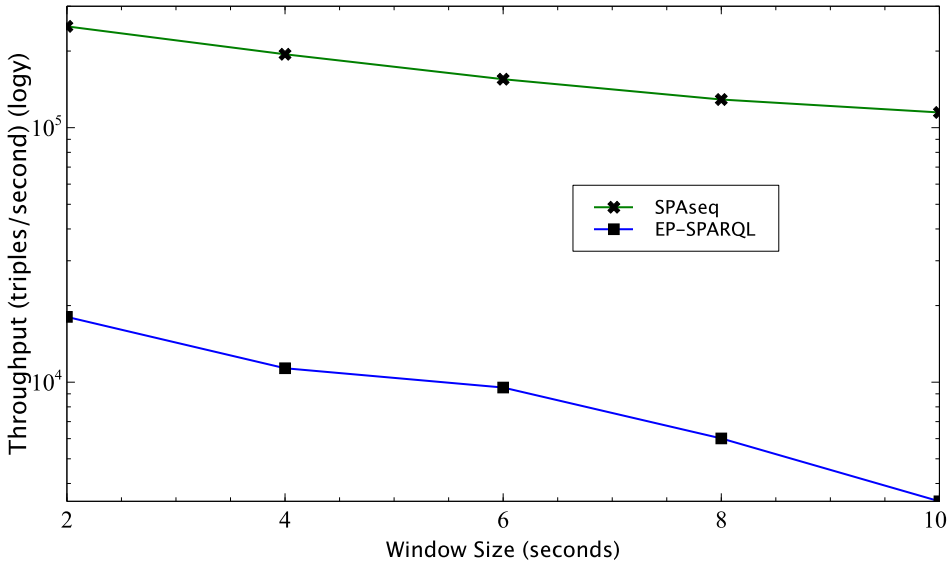

Figure 9.20: Comparative Analysis of SPAseq and EP-SPARQL over Variable Window Size

Figure 9.20 and 9.21 show the performance of both systems. From these results, we can see that, as expected, SPAseq yields much higher throughput compared with EP-SPARQL for both scenarios, i.e., increasing the window size and increasing the number of sequence patterns. Since, results provided in [LP+11] show that RSP systems such as CQELS outperforms ETALIS, which is the underlying engine for EP-SPARQL, in terms of performance and scalability; and our system SPECTRA, which is the underlying system of SPAseq outperforms other RSP systems. Hence, the complexity introduced by the temporal operators is well managed by our NFA_{scep} model and optimisation techniques. The performance of EP-SPARQL degrades quadratically with the increase of window size (Figure 9.20): EP-SPARQL uses a Prolog-wrapper based on the *event driven backward chaining rules* (EDBC), and schedules the execution via a declarative language using backward reasoning. This first results in an overhead of object mappings, second, reasoning with backward chaining is a complex and computing intensive task: it uses a goal-based memory management technique, i.e., periodic pruning of expired goals using alarm predicates, which is expensive for large windows. On the contrary, SPAseq employs NFA_{scep} model with various optimisation strategies to reduce the cost of triple patterns joins and the evaluation cost of the state-transition predicates. It utilises efficient "right-on-time" garbage collection for the deceased runs, and optimisations such

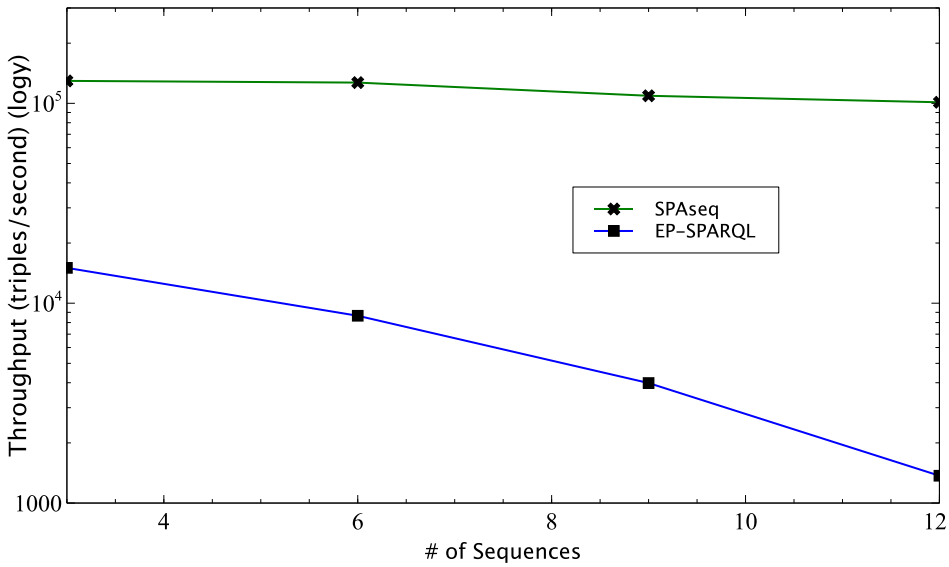

Figure 9.21: Comparative Analysis of SPAseq and EP-SPARQL over Variable # of Sequences

as pushing temporal windows and stateful joins, and incremental indexing from SPECTRA reduces the average computation overheads and life-span of an active run. In addition, the NFA-based executional model is much more immune to the increase in the number of sequence operators compared with EP-SPARQL: with the increase in sequence operators, the active life of each run also increases, however employing above mentioned optimisation techniques greatly reduces the life-span and the number of active runs.

9.6 Summary

How to build an optimised SCEP system? In this chapter, through the implementation details of the SPAseq query language, we covered such question. We started by providing the motivation behind the NFA-based engine for SPAseq. Later, we provided the NFA$_{scep}$ model, the compilation process of SPAseq queries, their evaluation and optimisation techniques. With the provided discussion in this chapter, we see that integrating RDF graph streams with an NFA model is not a straight-forward procedure and requires various customised implementation and optimisation techniques. Lastly, while utilising real-world and synthetic datasets we showcased the usability and performance of our system. In summary, our contributions for this chapter are as follows:

- **NFA$_{scep}$ Model for SPAseq Queries.** We presented the NFA$_{scep}$ model for SPAseq queries, where the GPM expressions are mapped on the state-transition predicates, and temporal operators result in a set of edges for each state.
- **Compilation of NFA$_{scep}$.** We detailed the compilation process of SPAseq queries with the NFA$_{scep}$ model.
- **Evaluation of NFA$_{scep}$.** We presented the evaluation process of NFA$_{scep}$, and examined how each operator is handled according to its defined attributes.
- **Evaluation Complexity of NFA$_{scep}$.** We provided the evaluation complexity of NFA$_{scep}$, and showed how different operators can be costly due to their inherent

nature.

- **Optimisation Techniques for NFA$_{scep}$.** We provided multiple optimisation techniques for the NFA$_{scep}$ evaluation. Some of these techniques are borrowed and customised from the CEP literature, and others are provided considering the streamset data model.
- **Experimental Evaluation of SPAseq queries.** We provided a detailed experimental analysis of the SPAseq queries. It gives a clear view of the costs of various temporal operators, and how the optimisation techniques affect the performance of the system. Furthermore, we provided a comparative analysis between EP-SPARQL and SPAseq queries.

The RDF graph model enables flexible representation of events to support the integration of heterogenous streams and interpretation of knowledge in a coherent manner. However, supporting SCEP over RDF graph streams first require an expressive query language, and second a scalable and optimised implementation of its processing model. In this chapter, we provided a complete framework for the SPAseq query language, which adheres to the aforementioned attributes, and our experimental analysis showed its practical importance.

Part IV

Conclusion and Future Perspectives

10 Conclusion

10.1 RDF Graph Stream Processing

In order to add and process the temporal nature of RDF graphs, our approaches leverage methods from graph summarisation, incremental query processing, vertical partitioning, and incremental evaluation. The combination of these techniques have enabled us to answer the following question.

> ***How can we provide a performance intensive framework to process RDF graph streams?***

We first contributed with SPECTRA, a framework that incrementally evaluates RDF graph streams, where each RDF graph event constitutes an RDF graph associated with a timestamp. It consists of three operators, namely SummaryGraph, QueryProc and IncrQueryProc. The SummaryGraph operator combines the structural and selectivity information of the defined query graph and extracts the useful triples from each RDF graph event. The QueyProc operator uses an incremental indexing technique, where the set of vertically partitioned tables are joined and indices – using sibling lists – are created during the join process. Thus, the creation of the indexing is part of the query process. Moreover, as hash-joins between views discard most of unwanted triples, indexing is performed on a smaller subsets of triples that can be the part of the final matched results. The IncQueryProc operator employs the already matched results and joins them with the newly arrived events. Therefore, the system takes the advantages of the already computed matches. In addition, the incremental indexing also provided a scalable way of removing the older triples from the defined window, i.e., without re-evaluating the matches. The combination of these operators proved to be quite useful in our experimental analysis. A series of experiments over real-world and synthetic datasets showed that our system outperforms state-of-the-art approaches by an order of magnitude.

10.2 Semantic Complex Event Processing

To make sense of the atomic RDF graph events, we focused on two main aspects of the SCEP: an expressive query language, and a scalable implementation of SCEP. We draw

out a set of expressive temporal operators for the existing CEP systems and integrated it with the core SPARQL operators. This resulted in an expressive language for SCEP over RDF graph streams. In addition, we provided a scalable implementation of our language with an NFA$_{scep}$ model. This enabled us to answer the following question.

How can we provide an expressive SCEP query language, and how can we accomplish its efficient implementation?

We contributed with SPAseq, a query language for processing RDF graph events with temporal operators. We provide the syntax and semantics of our language, and demonstrate its usefulness with various use cases. The strength of SPAseq is that it provides a clear separation between the operators for graph pattern matching over RDF graphs (GPM expressions) and the temporal operators over RDF graph events (sequence expression). This leads to a language that is easily extendable and independent of the underlying executional framework. SPAseq works on a streamset data model, thus multiple streams can be queried, and it offers multiple new operators that are not supported by existing languages. That is, kleene-+, explicit negation, event selection strategies, disjunction and conjunction over a set of events. Moreover, we provided a qualitative analysis of SPAseq and other SCEP languages that clearly showed its superiority.

We also contributed to the implementation of SPAseq. We introduced an NFA$_{scep}$ model, where the SPAseq operators are mapped on to the set of states as state-transition predicates. We showed how the GPM expressions are compiled using the standard SPARQL operators, and how the sequence expression results into a set of state-transition predicates. Based on this, we provided an efficient evaluation of NFA$_{scep}$ with multiple global and local-level optimisations. We used the lessons learned from SPECTRA, and integrated it as a GPM evaluator with the SPAseq query engine. Our optimisation techniques include: pushing the stateful joins, clustering of runs, and local optimisation for the disjunction and conjunction operators. The combination of these optimisations resulted in a scalable SCEP systems that can employ multiple heterogeneous RDF graph streams with expressive temporal operators. Using multiple real-world and synthetic datasets, we showed the usefulness of our optimisation techniques.

10.3 Impact

This work has a broad impact on a variety of applications: anomaly detection in dynamic graphs, event processing in various types of networks including social network, sensor network, transportation network, etc. Our research results have been published in a number of top-tier international conferences, and one of our paper [GPL16a] received an honourable mention award in ACM DEBS'16 conference.

11 Future Perspectives

The overarching theme of our research is developing scalable data structures and algorithms to understand the flow of RDF graph streams, and to extract knowledge out of them. In this work, we have taken several steps towards providing a scalable RDF stream processing framework and semantically-enabled complex event processing. Next we outline some of the research directions stemming from our work.

11.1 Top-k Operator over RDF Graph Streams

The *top-k* operator over streams monitors the incoming data items within a defined sliding window w to identify k highest-ranked data items. These data items are ranked with respect to a given *scoring function*. The main challenges faced by the top-k operator include: (i) maintaining a *candidate list* of data items that can be the part of top-k list, (ii) dealing with the non-appendable nature of real-world streams, where the arrival of a new data items may change the state of existing ones in top-k or candidate list. Due to the unbounded nature of the streams, all the data items cannot be stored in a sorted order and it requires efficient algorithms and data structures.

There has been no concrete work in the Semantic Web community that deals with top-k operator over RDF streams, while the relational-based solutions only work on append-able data streams. However, we claim that such an operator can easily be integrated over SPECTRA. As a proof, we implemented a customised version of a top-k operator for DEBS Grand Challenge 2015 [Gil+15]. Our solution employs a customised scoring function – as described in the DEBS Grand Challenge[1] – and uses the *New York Taxi* dataset (more details are provided in [Gil+15]). Each incoming RDF graph event is processed using the techniques described in SPECTRA, and a new tree-based data structure is employed to record the candidate list of RDF graphs. We call such tree a *Range Tree* [Gil+15], where the RDF graphs in candidate list are compressed using the range defined in each of the tree nodes. This results in a scalable solution, where hundreds of thousands of RDF graphs can be stored in a tree, with each node containing a handful of graphs with similar

[1]DEBS Grand Challenge 2015: `http://www.debs2015.org/call-grand-challenge.html`, last accessed: July, 2016.

scoring function. We claim that such solution can easily be generalised, where the user can define the scoring function in a high-level query. Our future perspectives in this direction are to define an adaptable range tree customised according to the user defined functions.

11.2 Multicore Mode for the RDF Graph Streams

Computer architectures are increasingly based on multicore CPU's with large main-memories. Hence, it would be beneficial for an RDF graph stream processing system to effectively exploit the increasing amount of DRAM and multicore processor. One step in this direction would be to build a customised and pluggable scheduling framework for SPECTRA and SPAseq. Such framework would parallelise the execution of specific query operators on specific application-provided threads or cores. Hence, taking the physical plan, it would partition it into query fragments/operators. Given, n threads, each thread picks up data batches to be evaluated on the selected operators, albeit the temporal order needs to be considered in this setting. The design of SPAseq permits us to divide the main query into a set of query fragments, where each priority queue holds a query fragment. Next the scheduler thread picks up the fragment with the highest priority to be executed. Such a framework on top of the SPECTRA would boost its performance, and would enable it to fully utilise the available cores in modern CPUs.

11.3 Processing RDF Graph Streams in Distributed Environments

The past decade has witnessed an increasing adoption of cloud technology, which provides better scalability, availability, and fault-tolerance via transparent partitioning and replication, and automatic load balancing. We believe that the fate of RDF graph stream processing and SCEP lies in accessing the distributed resource, especially in the so-called cloud. In this thesis, we provided an insight into the scalable in-memory solutions of RDF graph streams, where events are partitioned into a set of vertically partitioned tables, and the SCEP query language has a clear separation of constructs. These insights can be harnessed to provide a scale-out solution for the problems discussed in this thesis. As a preliminary work, recently we proposed a framework called DIONYSUS[2] [GPL16b]: it is motivated by the following goals (more details can be found in [GPL16b]).

- We are interested in an efficient distribution of streaming data from a set of sources, which are not known in advance. Thus, our storage and data distribution model is envisioned by *Common Basic Graph Pattern store* (CBGP-store). Each CBGP-store is assigned with a generic BGP (i.e., a set of triple patterns) generated automatically/manually from the domain ontology and domain use cases. A collection of such stores exposes (i) fresh incrementally computed results of a set of CBGP for streaming queries, (ii) the set of previously computed results for off-line analytical queries. This enables the use of incremental indexing techniques to efficiently store data for each CBGP (see Figure 11.1).
- The second goal of our approach is to push the intensive query optimisation and processing locally at each CBGP-store for *Exact Query Graphs* (EQGs) registered by a user. An EQG is a more selective form of CBGP and it contains (i) subsets of triple

[2] **D**istr**I**buted a**N**al**Y**tical, **S**treaming and **S**equence q**U**erie**S**

patterns that are distributed among CBGP-stores, (ii) SPARQL query operators, such as `select, optional, union, filter, group by`, etc. The results of each EQGs are accumulated at the federated level.

- Our third goal is to enable different kinds of queries – such as analytical, streaming, sequence-based – through a single query interface. A query interface encompasses subsets of CBGP-stores that can be abstracted as *islands of CBGP-stores* (see Figure 11.1). This would enable to share query results computation and local optimisation strategies. For example, users would like to get the result of (i) an analytical query describing the number of active appliances and their power usage in a house, and the result of (ii) a sequence-based query to determine the sequence of power usage by various house appliances. This calls for a single query language, where each query operator is optimised according to its defined characteristics.
- Our fourth aim would be to provide the *semantic completeness* and *locations transparency*. That is, a new source can be added without affecting the integrity constraints, and a user query can span multiple islands of CBGP-stores. This would enable us to first, share the optimising strategies defined for each CBGP-store, second reducing the network traffic by employing local optimisation and computation strategies.

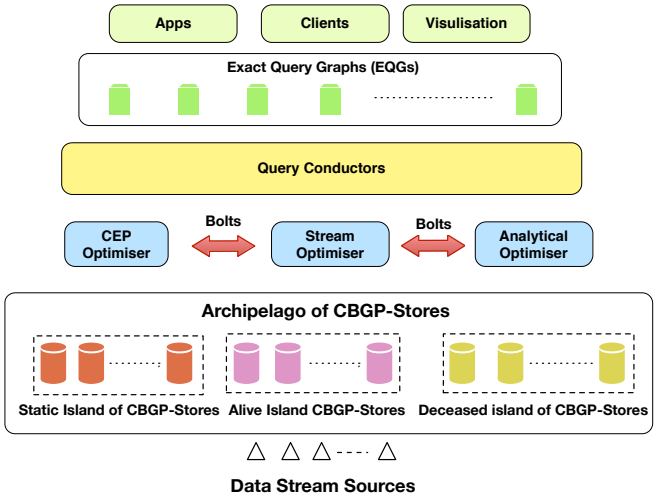

Figure 11.1: Layered Architecture of DIONYSUS

In summary, our envisioned system can provide a way not to drown in the sea of information emanating from heterogeneous distributed sources. It filters unnecessary information, which otherwise can result in excessive use of storage and computational resources. The framework is designed to minimise the burden of query evaluation at the federation layer and to share local optimisation strategies across the islands of CBGP-stores.

In conclusion, scalable and performance intensive RDF stream processing and semantically-enabled complex event processing have numerous high-impact applications, and fascinating research challenges.

Appendices

A Dataset Queries

This appendix describes the set of queries that we have utilised for our empirical evaluations in Chapter 7 and Chapter 9.

A.1 LUBM Queries

```
prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>
prefix xsd: <http://www.w3.org/2001/XMLSchema#>
prefix ub: <http://swat.cse.lehigh.edu/onto/univ-bench.owl#>
prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT ?y ?z ?x WHERE
{
?z ub:subOrganizationOf ?y .
?x ub:memberOf ?z .
?x ub:undergraduateDegreeFrom ?y .
?x rdf:type ub:GraduateStudent .
?y rdf:type ub:University .
?z rdf:type ub:Department .
}
```

Query A.1: LUBM-Q1

```
prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>
prefix xsd: <http://www.w3.org/2001/XMLSchema#>
prefix ub: <http://swat.cse.lehigh.edu/onto/univ-bench.owl#>
prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT ?y ?x WHERE
{
?x rdf:type ub:Course .
?x ub:name ?y .
}
```

Query A.2: LUBM-Q2

```
prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>
prefix xsd: <http://www.w3.org/2001/XMLSchema#>
prefix ub: <http://swat.cse.lehigh.edu/onto/univ-bench.owl#>
```

```
prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT ?y ?x ?z WHERE
{ ?x ub:memberOf ?z .
?z ub:subOrganizationOf ?y .
?x ub:undergraduateDegreeFrom ?y .
?x rdf:type ub:UndergraduateStudent.
?y rdf:type ub:University .
?z rdf:type ub:Department .
}
```

Query A.3: LUBM-Q3

```
prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>
prefix xsd: <http://www.w3.org/2001/XMLSchema#>
prefix ub: <http://swat.cse.lehigh.edu/onto/univ-bench.owl#>
prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT ?x ?y1 WHERE
{ ?x ub:worksFor ?k .
?x ub:name ?y1 .
?x ub:emailAddress ?y2 .
?x ub:telephone ?y3 .
?x rdf:type ub:FullProfessor .
Filter(?k = <http://www.Department1.University0.edu> ) }
```

Query A.4: LUBM-Q4

```
prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>
prefix xsd: <http://www.w3.org/2001/XMLSchema#>
prefix ub: <http://swat.cse.lehigh.edu/onto/univ-bench.owl#>
prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT ?x WHERE
{ ?x ub:subOrganizationOf ?k .
?x rdf:type ub:ResearchGroup .
Filter (?k = <http://www.Department1.University0.edu>) }
```

Query A.5: LUBM-Q5

```
prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>
prefix xsd: <http://www.w3.org/2001/XMLSchema#>
prefix ub: <http://swat.cse.lehigh.edu/onto/univ-bench.owl#>
prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT ?x ?y WHERE
{ ?x ub:worksFor ?y .
?y ub:subOrganizationOf ?k.
?y rdf:type ub:Department .
?x rdf:type ub:FullProfessor .
Filter (?k = <http://www.University0.edu<) }
```

Query A.6: LUBM-Q6

```
prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>
prefix xsd: <http://www.w3.org/2001/XMLSchema#>
prefix ub: <http://swat.cse.lehigh.edu/onto/univ-bench.owl#>
prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT ?x ?y ?z WHERE
{
?y ub:teacherOf ?z .
?x ub:takesCourse ?z.
?x ub:advisor ?y .
```

```
?z rdf:type ub:Course .
?x rdf:type ub:UndergraduateStudent .
?y rdf:type ub:FullProfessor .
}
```

Query A.7: LUBM-Q7

A.2 SNB Queries

```
SELECT ?post ?creator ?loc
           WHERE { ?post rdf:type snvoc:Post.
                    ?post snvoc:id ?id.
                    ?post snvoc:creationDate ?date.
                    ?post snvoc:hasCreator ?creator.
                    ?post snvoc:hasTag ?tag.
                    ?post snvoc:isLocatedIn ?loc.
                     }
```

Query A.8: SNB-Q1

```
SELECT ?forum ?meb ?interest WHERE
                    { ?forum snvoc:title ?title.
                     ?forum snvoc:hasMember ?meb.
                     ?meb   snvoc:hasPerson ?p.
                     ?p  snvoc:hasInterest ?interest.
                     }
```

Query A.9: SNB-Q2

```
SELECT ?post ?creator ?id WHERE
                     {
                     ?cmnt snvoc:hasCreator ?creator.
                     ?creator snvoc:speaks ?lang.
                     ?cmnt snvoc:isLocatedIn  <http://dbpedia.org/resource
   /Nicaragua>.
                     ?cmnt snvoc:replyOf ?post.
                     ?post snvoc:hasCreator  ?id. }
```

Query A.10: SNB-Q3

A.3 LSBench Queries

```
SELECT ?post1 ?user ?post2
WHERE {
?post1 snvoc:content ?cont.
?post1 snvoc:hasCreator ?user.
?post2 snvoc:content ?cont2.
?post2 snvoc:hasCreator ?user.
}
```

Query A.11: LS-Q1

```
SELECT ?post ?person1 ?person2
WHERE {
    ?person1  snvoc:likes     ?p.
    ?p        snvoc:hasPost   ?post.
    ?p2       snvoc:hasPost   ?post.
    ?person2  snvoc:likes     ?p2.
    ?person1  snvoc:knows     ?know.
    ?know     snvoc:hasPerson ?person2.
}
```

Query A.12: LS-Q2

```
SELECT ?post1 ?user1 ?user2
WHERE {
    ?post1 snvoc:hasCreator  ?user.
    ?post1 snvoc:content     ?cont.
    ?know  snvoc:hasPerson   ?user.
    ?user2 snvoc:knows       ?know.
    ?user2 snvoc:hasInterest
        <http://dbpedia.org/resource/Charles_Dickens>.
}
```

Query A.13: LS-Q3

A.4 SEAS Queries

```
prefix seas: <http://purl.org/NET/seas#>
prefix m: <http://purl.org/NET/seas/measures#>
SELECT * WHERE
{
?obs m:V_RMS ?rm.
?obs m:V_THD ?thd.
?obs m:V_CF ?cf.
?obs m:W ?watt.
?obs m:Wh ?watth.
?obs m:DPF ?vah.
}
```

Query A.14: SEAS-Q1

```
prefix seas: <http://purl.org/NET/seas#>
prefix m: <http://purl.org/NET/seas/measures#>
SELECT * WHERE
{
?obs m:V_RMS ?rm.
?obs m:V_THD ?thd.
?obs m:V_CF ?cf.
?obs m:W ?watt.
?obs m:Wh ?watth.
?obs m:DPF ?vah.
Filter( ?watt > 10 && ?cf < 1.2) }
```

Query A.15: SEAS-Q2

A.5 V-Shaped Pattern Queries for SPAseq and EP-SPARQL

A.5.1 SPAseq Queries

```
prefix c: <http://example/company#>
prefix pred: <http://example/>
SELECT ?company ?p1 ?p2 ?p3 ?v1 ?v2 ?v3
WITHIN 10 SECONDS
FROM STREAM S1 <http://example.org/main>
WHERE
SEQ (A, B, C)
{
DEFINE GPM A ON S1

{
?company pred:price ?p1.
?company pred:volume ?vol1.
}
DEFINE GPM B ON S1
{
  ?company pred:price ?p2.
  ?company pred:volume ?vol2.
  Filter (?p2 < ?p1 )
}
DEFINE GPM C ON S1
{
  ?company pred:price ?p3.
  ?company pred:volume ?vol3.
  Filter (?p3 > ?p2 && ?p3 > ?p1).
 }
}
```

Query A.16: Vshape-Q1 with three GPM expressions

```
prefix c: <http://example/company#>
prefix pred: <http://example/>
SELECT ?company ?p1 ?p2 ?p3 ?p4 ?p5 ?p6  ?v1 ?v2 ?v3 ?v4 ?v5 ?v6
WITHIN 10 SECONDS
FROM STREAM S1 <http://example.org/main>
WHERE
SEQ (A, B, C, D, E, F)
{
DEFINE GPM A ON S1
 {
 ?company pred:price ?p1.
 ?company pred:volume ?vol1.
 }
DEFINE GPM B ON S1
 {
 ?company pred:price ?p2.
 ?company pred:volume ?vol2.
  Filter (?p2 < ?p1 )
 }
DEFINE GPM C ON S1
 {
 ?company pred:price ?p3.
 ?company pred:volume ?vol3.
 Filter (?p3 > ?p2 && ?p3 > ?p1).
 }
```

```
 DEFINE GPM D ON S1
 {
 ?company pred:price ?p4.
 ?company pred:volume ?vol4.
 Filter (?p4 < ?p3).
 }
 DEFINE GPM E ON S1
 {
 ?company pred:price ?p5.
 ?company pred:volume ?vol5.
  Filter (?p5 < ?p4).
 }
 DEFINE GPM F ON S1
 {
 ?company pred:price ?p6.
 ?company pred:volume ?vol6.
  Filter (?p6 > ?p5 && ?p6 > ?p4).
 }
}
```

Query A.17: Vshape-Q1 with six GPM expressions

The SPAseq queries with 9 and 12 GPM expressions can easily be inferred from the above mentioned queries.

A.5.2 EP-SPARQL V-Shaped Pattern

```
SELECT ?company ?p1 ?p2 ?p3 ?v1 ?v2 ?v3
WHERE
SEQ {?company price ?p1  EQUALS {?company volume ?v1 }  }

SEQ {?company price ?p2  EQUALS {?company volume ?v2 }  }

SEQ {?company price ?p3  EQUALS {?company volume ?v3 }  }

 Filter (?p2 < ?p1 && ?p3 > ?p2 && ?p3 > ?p1 && getDURATION() < "P10S"^^
    xsd:duration )
```

Query A.18: EP-SPARQL Query with three Sequence expressions

```
SELECT ?company ?p1 ?p2 ?p3 ?p4 ?p5 ?p6 ?v1 ?v2 ?v3 ?v4 ?v5 ?v6
WHERE
SEQ {?company price ?p1  EQUALS {?company volume ?v1 }  }

SEQ {?company price ?p2  EQUALS {?company volume ?v2 }  }

SEQ {?company price ?p3  EQUALS {?company volume ?v3 }  }

SEQ {?company price ?p4  EQUALS {?company volume ?v4 }  }

SEQ {?company price ?p5  EQUALS {?company volume ?v5 }  }

SEQ {?company price ?p6  EQUALS {?company volume ?v6 }  }

 Filter (?p2 < ?p1 && ?p3 > ?p2 && ?p3 > ?p1 && ?p4 < ?p3 && ?p5 < ?p4 &&
    ?p6 > ?p5 && ?p6 > ?p4 && getDURATION() < "P10S"^^xsd:duration )
```

Query A.19: EP-SPARQL Query with three Sequence expressions

B

List of Related Publications

- Syed Gillani, Frédérique Laforest, and Gauthier Picard. "A Generic Ontology for Prosumer-Oriented Smart Grid". In: Proceedings of the Workshops of the EDBT/ICDT 2014 Joint Conference (EDBT/ICDT 2014), Athens, Greece, 2014.

- Syed Gillani, Frédérique Laforest, and Gauthier Picard. "Towards Efficient Semantically Enriched Complex Event Processing and Pattern Matching". In: Proceedings of the 3rd International Workshop on Ordering and Reasoning Co-located with the 13th International Semantic Web Conference (ISWC 2014), Italy, 2014.

- Syed Gillani, Gauthier Picard, and Frédérique Laforest. "IntelSCEP: Towards an Intelligent Semantic Complex Event Processing Framework for Prosumer- Oriented SmartGrid". In: Proceedings of the 2014 International Workshop on Web Intelligence and Smart Sensing. IWWISS '14. Saint Etienne, France, 2014.

- Syed Gillani, Gauthier Picard, and Frédérique Laforest. "Continuous Graph Pattern Matching over Knowledge Graph Streams". In: Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems. DEBS '16. Irvine, California USA, 2016.

- Syed Gillani, Gauthier Picard, and Frédérique Laforest. "SPECTRA: Continuous Query Processing for RDF Graph Streams Over Sliding Windows". In: Proceedings of the 28th International Conference on Scientific and Statistical Database Management. SSDBM '16. Budapest, Hungary, 2016.

- Syed Gillani, Abderrahmen Kammoun, Julian Subercaze, Kamal Singh, Gauthier Picard, and Frédérique Laforest . "Top-K Queries in RDF Graph-based Stream Processing with Actors". In: Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems. DEBS '15. Oslo, Norway, 2015.

- Abderrahmen Kammoun, Syed Gillani, Julian Subercaze and Christophe Gravier. "High Performance Top-k Processing of Non-linear Windows over Data Streams". In: Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems. DEBS '16. Irvine, California USA, 2016.

- Syed Gillani, Gauthier Picard, and Frédérique Laforest. "DIONYSUS: Towards Query-aware Distributed Processing of RDF Graph Streams.". In: Proceedings of the Workshops (GraphQ) of the EDBT/ICDT 2016 Joint Conference (EDBT/ICDT 2016), Bordeaux, France, 2016.

Bibliography

[Aba+03] Daniel J. Abadi et al. “Aurora: A New Model and Architecture for Data Stream Management”. In: *The VLDB Journal* 12.2 (Aug. 2003), pp. 120–139. ISSN: 1066-8888. DOI: 10.1007/s00778-003-0095-z. URL: http://dx.doi.org/10.1007/s00778-003-0095-z.

[Aba+07] Daniel J. Abadi et al. “Scalable Semantic Web Data Management Using Vertical Partitioning”. In: *Proceedings of the 33rd International Conference on Very Large Data Bases*. VLDB ’07. Vienna, Austria: VLDB Endowment, 2007, pp. 411–422. ISBN: 978-1-59593-649-3. URL: http://dl.acm.org/citation.cfm?id=1325851.1325900.

[Aba+09a] Daniel J. Abadi et al. “SW-Store: A Vertically Partitioned DBMS for Semantic Web Data Management”. In: *The VLDB Journal* 18.2 (Apr. 2009), pp. 385–406. ISSN: 1066-8888. DOI: 10.1007/s00778-008-0125-y. URL: http://dx.doi.org/10.1007/s00778-008-0125-y.

[Aba+09b] Daniel J. Abadi et al. “SW-Store: A Vertically Partitioned DBMS for Semantic Web Data Management”. In: *The VLDB Journal* 18.2 (Apr. 2009), pp. 385–406. ISSN: 1066-8888. DOI: 10.1007/s00778-008-0125-y. URL: http://dx.doi.org/10.1007/s00778-008-0125-y.

[ABW06] Arvind Arasu, Shivnath Babu, and Jennifer Widom. “The CQL Continuous Query Language: Semantic Foundations and Query Execution”. In: *The VLDB Journal* 15.2 (June 2006), pp. 121–142. ISSN: 1066-8888. DOI: 10.1007/s00778-004-0147-z. URL: http://dx.doi.org/10.1007/s00778-004-0147-z.

[AC06] Raman Adaikkalavan and Sharma Chakravarthy. “SnoopIB: Interval-based Event Specification and Detection for Active Databases”. In: *Data Knowl. Eng.* 59.1 (Oct. 2006), pp. 139–165. ISSN: 0169-023X. DOI: 10.1016/j.datak.2005.07.009. URL: http://dx.doi.org/10.1016/j.datak.2005.07.009.

[AC10] Medha Atre and Chaoji. “Matrix "Bit" Loaded: A Scalable Lightweight Join Query Processor for RDF Data”. In: *WWW*. 2010, pp. 41–50. DOI: 10.1145/1772690.1772696.

[AF11] Mario Arias and Javier D. Fernández. “An Empirical Study of Real-World SPARQL Queries”. In: *CoRR* abs/1103.5043 (2011). URL: http://arxiv.org/abs/1103.5043.

[Agr+08] Jagrati Agrawal et al. “Efficient Pattern Matching over Event Streams”. In: *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*. SIGMOD ’08. Vancouver, Canada: ACM, 2008, pp. 147–160. ISBN: 978-1-60558-102-6. DOI: 10.1145/1376616.1376634. URL: http://doi.acm.org/10.1145/1376616.1376634.

[AH00] Ron Avnur and Joseph M. Hellerstein. "Eddies: Continuously Adaptive Query Processing". In: *SIGMOD Rec.* 29.2 (May 2000), pp. 261–272. ISSN: 0163-5808. DOI: 10.1145/335191.335420. URL: http://doi.acm.org/10.1145/335191.335420.

[Ani+10] Darko Anicic et al. "Web Reasoning and Rule Systems: Fourth International Conference, RR 2010, Bressanone/Brixen, Italy, September 22-24, 2010. Proceedings". In: ed. by Pascal Hitzler and Thomas Lukasiewicz. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010. Chap. A Rule-Based Language for Complex Event Processing and Reasoning, pp. 42–57. ISBN: 978-3-642-15918-3. DOI: 10.1007/978-3-642-15918-3_5. URL: http://dx.doi.org/10.1007/978-3-642-15918-3_5.

[Ani+11] Darko Anicic et al. "EP-SPARQL: A Unified Language for Event Processing and Stream Reasoning". In: *Proceedings of the 20th International Conference on World Wide Web*. WWW '11. Hyderabad, India: ACM, 2011, pp. 635–644. ISBN: 978-1-4503-0632-4. DOI: 10.1145/1963405.1963495. URL: http://doi.acm.org/10.1145/1963405.1963495.

[Ani+12] Darko Anicic et al. "Stream Reasoning and Complex Event Processing in ETALIS". In: *Semant. web* 3.4 (Oct. 2012), pp. 397–407. ISSN: 1570-0844. URL: http://dl.acm.org/citation.cfm?id=2590208.2590214.

[Ara+04] A. Arasu et al. *STREAM: The Stanford Data Stream Management System*. Technical Report 2004-20. Stanford InfoLab, 2004. URL: http://ilpubs.stanford.edu:8090/641/.

[Atr+10] Medha Atre et al. "Matrix "Bit" Loaded: A Scalable Lightweight Join Query Processor for RDF Data". In: *Proceedings of the 19th International Conference on World Wide Web*. WWW '10. Raleigh, North Carolina, USA: ACM, 2010, pp. 41–50. ISBN: 978-1-60558-799-8. DOI: 10.1145/1772690.1772696. URL: http://doi.acm.org/10.1145/1772690.1772696.

[Bab+02] Brian Babcock et al. "Models and Issues in Data Stream Systems". In: *Proceedings of the Twenty-first ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*. PODS '02. Madison, Wisconsin: ACM, 2002, pp. 1–16. ISBN: 1-58113-507-6. DOI: 10.1145/543613.543615. URL: http://doi.acm.org/10.1145/543613.543615.

[Bab+05] Shivnath Babu et al. "Adaptive caching for continuous queries". In: *21st International Conference on Data Engineering (ICDE'05)*. 2005, pp. 118–129. DOI: 10.1109/ICDE.2005.15.

[Bar+10a] Davide Francesco Barbieri et al. "An Execution Environment for C-SPARQL Queries". In: *Proceedings of the 13th International Conference on Extending Database Technology*. EDBT '10. Lausanne, Switzerland: ACM, 2010, pp. 441–452. ISBN: 978-1-60558-945-9. DOI: 10.1145/1739041.1739095. URL: http://doi.acm.org/10.1145/1739041.1739095.

[Bar+10b] Davide Francesco Barbieri et al. "Querying RDF Streams with C-SPARQL". In: *SIGMOD Rec.* 39.1 (Sept. 2010), pp. 20–26. ISSN: 0163-5808. DOI: 10.1145/1860702.1860705. URL: http://doi.acm.org/10.1145/1860702.1860705.

[Baz+15] Hamid R. Bazoobandi et al. "A Compact In-Memory Dictionary for RDF Data". In: *ESWC*. 2015, pp. 205–220.

[BC08] Michela Becchi and Patrick Crowley. "Efficient Regular Expression Evaluation: Theory to Practice". In: *Proceedings of the 4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*. ANCS '08. San Jose, California: ACM, 2008, pp. 50–59. ISBN: 978-1-60558-346-4. DOI: 10.1145/1477942.1477950. URL: http://doi.acm.org/10.1145/1477942.1477950.

[BCM06] Roger S. Barga and Hillary Caituiro-Monge. "Event Correlation and Pattern Detection in CEDR." In: *EDBT Workshops*. Ed. by Torsten Grust et al. Vol. 4254. Lecture Notes in Computer Science. Springer, Nov. 13, 2006, pp. 919–930. ISBN: 3-540-46788-2. URL: http://dblp.uni-trier.de/db/conf/edbtw/edbtw2006.html#BargaC06.

[BDM07] Brian Babcock, Mayur Datar, and Rajeev Motwani. "Data Streams: Models and Algorithms". In: ed. by Charu C. Aggarwal. Boston, MA: Springer US, 2007. Chap. Load Shedding in Data Stream Systems, pp. 127–147. ISBN: 978-0-387-47534-9. DOI: 10.1007/978-0-387-47534-9_7. URL: http://dx.doi.org/10.1007/978-0-387-47534-9_7.

[BE07] François Bry and Michael Eckert. "Rule-Based Composite Event Queries: The Language XChangeEQ and its Semantics". In: *Proceedings of First International Conference on Web Reasoning and Rule Systems, Innsbruck, Austria (7th–8th June 2007)*. Vol. 4524. LNCS. 2007. URL: http://www.pms.ifi.lmu.de/publikationen/#PMS-FB-2007-4.

[BGJ08] Andre Bolles, Marco Grawunder, and Jonas Jacobi. "Streaming SPARQL - Extending SPARQL to Process Data Streams". In: *The Semantic Web: Research and Applications: 5th European Semantic Web Conference, ESWC 2008, Tenerife, Canary Islands, Spain, June 1-5, 2008 Proceedings*. Ed. by Sean Bechhofer et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 448–462. ISBN: 978-3-540-68234-9. DOI: 10.1007/978-3-540-68234-9_34. URL: http://dx.doi.org/10.1007/978-3-540-68234-9_34.

[Bie+14] Meghyn Bienvenu et al. "Ontology-Based Data Access: A Study Through Disjunctive Datalog, CSP, and MMSNP". In: *ACM Trans. Database Syst.* 39.4 (Dec. 2014), 33:1–33:44. ISSN: 0362-5915. DOI: 10.1145/2661643. URL: http://doi.acm.org/10.1145/2661643.

[Biz+09] Christian Bizer et al. "DBpedia - A Crystallization Point for the Web of Data". In: *Web Semant.* 7.3 (Sept. 2009), pp. 154–165. ISSN: 1570-8268. DOI: 10.1016/j.websem.2009.07.002. URL: http://dx.doi.org/10.1016/j.websem.2009.07.002.

[BKH02] Jeen Broekstra, Arjohn Kampman, and Frank van Harmelen. "Sesame: A Generic Architecture for Storing and Querying RDF and RDF Schema". In: *Proceedings of the First International Semantic Web Conference on The Semantic Web*. ISWC '02. London, UK, UK: Springer-Verlag, 2002, pp. 54–68. ISBN: 3-540-43760-6. URL: http://dl.acm.org/citation.cfm?id=646996.711426.

[BL06] T Berners-Lee. "Linked Data. W3C Design Issues," in: *Technical report W3C*. 2006. URL: https://www.w3.org/DesignIssues/LinkedData.html.

[BL80] T Berners-Lee. "The ENQUIRE System – Short Description (1.1)." In: *Technical report, European Organisation for Nuclear Research, 1980*. 1980. URL: http://www.w3.org/History/1980/Enquire/manual/.

[BL93] T Berners-Lee. "A Brief History of the Web. W3C Design Issues, 1993". In: *Online Article*. 1993. URL: http://www.w3.org/DesignIssues/TimBook-old/History.html.

[BL98] T Berners-Lee. "Semantic Web Road map. 1998". In: *Online Article*. 1998. URL: http://www.w3.org/DesignIssues/TimBook-old/History.html.

[Bot+10] I. Botan et al. "SECRET: A Model for Analysis of the Execution Semantics of Stream Processing Systems". In: *International Conference on Very Large Data Bases (VLDB'10)*. Singapore, 2010.

[Bra+16] Vladimir Braverman et al. "Clustering Problems on Sliding Windows". In: *Proceedings of the Twenty-Seventh Annual ACM-SIAM Symposium on Discrete Algorithms*. SODA '16. Arlington, Virginia: Society for Industrial and Applied Mathematics, 2016, pp. 1374–1390. ISBN: 978-1-611974-33-1. URL: http://dl.acm.org/citation.cfm?id=2884435.2884530.

[Bre+07] Lars Brenna et al. "Cayuga: A High-performance Event Processing Engine". In: *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*. SIGMOD '07. Beijing, China: ACM, 2007, pp. 1100–1102. ISBN: 978-1-59593-686-8. DOI: 10.1145/1247480.1247620. URL: http://doi.acm.org/10.1145/1247480.1247620.

[Bre+09] Lars Brenna et al. "Distributed Event Stream Processing with Non-deterministic Finite Automata". In: *Proceedings of the Third ACM International Conference on Distributed Event-Based Systems*. DEBS '09. Nashville, Tennessee: ACM, 2009, 3:1–3:12. ISBN: 978-1-60558-665-6. DOI: 10.1145/1619258.1619263. URL: http://doi.acm.org/10.1145/1619258.1619263.

[Bry+07] François Bry et al. "Evolution of Distributed Web Data: An Application of the Reactive Language XChange". In: *Proceedings of IEEE 23rd International Conference on Data Engeneering, Istanbul, Turkey (15th–20th April 2007)*. 2007. URL: http://www.pms.ifi.lmu.de/publikationen/#PMS-FB-2007-3.

[BSS96] Michael H. Böhlen, Richard Thomas Snodgrass, and Michael D. Soo. "Coalescing in Temporal Databases". In: *Proceedings of the 22th International Conference on Very Large Data Bases*. VLDB '96. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1996, pp. 180–191. ISBN: 1-55860-382-4. URL: http://dl.acm.org/citation.cfm?id=645922.673474.

[BV10] Thomas BERNHARDT and Alexandre VASSEUR. "ESPER-complex event processing," in: *Online Article*. 2010. URL: http://www.espertech.com/products/esper.php.

[Cal+07] Diego Calvanese et al. "Tractable Reasoning and Efficient Query Answering in Description Logics: The DL-Lite Family". In: *J. Autom. Reason.* 39.3 (Oct. 2007), pp. 385–429. ISSN: 0168-7433. DOI: 10.1007/s10817-007-9078-x. URL: http://dx.doi.org/10.1007/s10817-007-9078-x.

[Car+04] Jeremy J. Carroll et al. "Jena: Implementing the Semantic Web Recommendations". In: *Proceedings of the 13th International World Wide Web Conference on Alternate Track Papers &Amp; Posters*. WWW Alt. '04. New York, NY, USA: ACM, 2004, pp. 74–83. ISBN: 1-58113-912-8. DOI: 10.1145/1013367.1013381. URL: http://doi.acm.org/10.1145/1013367.1013381.

[CCG10] Jean-Paul Calbimonte, Oscar Corcho, and Alasdair J. G. Gray. "Enabling Ontology-based Access to Streaming Data Sources". In: *Proceedings of the 9th International Semantic Web Conference on The Semantic Web - Volume Part I*. ISWC'10. Shanghai, China: Springer-Verlag, 2010, pp. 96–111. ISBN: 3-642-17745-X, 978-3-642-17745-3. URL: http://dl.acm.org/citation.cfm?id=1940281.1940289.

[CFT08] Kendall Grant Clark, Lee Feigenbaum, and Elias Torres. *SPARQL Protocol for RDF*. World Wide Web Consortium, Recommendation. 2008.

[Cha+03] Sirish Chandrasekaran et al. "TelegraphCQ: Continuous Dataflow Processing". In: *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*. SIGMOD '03. San Diego, California: ACM, 2003, pp. 668–668. ISBN: 1-58113-634-X. DOI: 10.1145/872757.872857. URL: http://doi.acm.org/10.1145/872757.872857.

[Cha+94] Sharma Chakravarthy et al. "Composite Events for Active Databases: Semantics, Contexts and Detection". In: *Proceedings of the 20th International Conference on Very Large Data Bases*. VLDB '94. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1994, pp. 606–617. ISBN: 1-55860-153-8. URL: http://dl.acm.org/citation.cfm?id=645920.672994.

[Cha+95] Sharma Chakravarthy et al. "ECA Rule Integration into an OODBMS: Architecture and Implementation". In: *Proceedings of the Eleventh International Conference on Data Engineering*. ICDE '95. Washington, DC, USA: IEEE Computer Society, 1995, pp. 341–348. ISBN: 0-8186-6910-1. URL: http://dl.acm.org/citation.cfm?id=645480.655427.

[Cha97] S. Chakravarthy. "Sentinel: An Object-oriented DBMS with Event-based Rules". In: *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data*. SIGMOD '97. Tucson, Arizona, USA: ACM, 1997, pp. 572–575. ISBN: 0-89791-911-4. DOI: 10.1145/253260.253409. URL: http://doi.acm.org/10.1145/253260.253409.

[CL04] Jan Carlson and Björn Lisper. "An Event Detection Algebra for Reactive Systems". In: *Proceedings of the 4th ACM International Conference on Embedded Software*. EMSOFT '04. Pisa, Italy: ACM, 2004, pp. 147–154. ISBN: 1-58113-860-1. DOI: 10.1145/1017753.1017779. URL: http://doi.acm.org/10.1145/1017753.1017779.

[CM12] Gianpaolo Cugola and Alessandro Margara. "Processing Flows of Information: From Data Stream to Complex Event Processing". In: *ACM Comput. Surv.* 44.3 (June 2012), 15:1–15:62. ISSN: 0360-0300. DOI: 10.1145/2187671.2187677. URL: http://doi.acm.org/10.1145/2187671.2187677.

[CMC16] Jean-Paul Calbimonte, José Mora, and Óscar Corcho. "Query Rewriting in RDF Stream Processing". In: *The Semantic Web. Latest Advances and New Domains - 13th International Conference, ESWC 2016, Heraklion, Crete, Greece, May 29 - June 2, 2016, Proceedings*. 2016, pp. 486–502. DOI: 10.1007/978-3-319-34129-3_30. URL: http://dx.doi.org/10.1007/978-3-319-34129-3_30.

[CN07] Surajit Chaudhuri and Vivek Narasayya. "Self-tuning Database Systems: A Decade of Progress". In: *Proceedings of the 33rd International Conference on Very Large Data Bases*. VLDB '07. Vienna, Austria: VLDB Endowment, 2007, pp. 3–14. ISBN: 978-1-59593-649-3. URL: http://dl.acm.org/citation.cfm?id=1325851.1325856.

[Cor11] Graham Cormode. "Sketch techniques for approximate query processing". In: *Synposes for Approximate Query Processing: Samples, Histograms, Wavelets and Sketches, Foundations and Trends in Databases*. 2011.

[Cra+02] Chuck Cranor et al. "Gigascope: High Performance Network Monitoring with an SQL Interface". In: *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*. SIGMOD '02. Madison, Wisconsin: ACM, 2002, pp. 623–623. ISBN: 1-58113-497-5. DOI: 10.1145/564691.564777. URL: http://doi.acm.org/10.1145/564691.564777.

[Cra+03] Chuck Cranor et al. "Gigascope: A Stream Database for Network Applications". In: *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*. SIGMOD '03. San Diego, California: ACM, 2003, pp. 647–651. ISBN: 1-58113-634-X. DOI: 10.1145/872757.872838. URL: http://doi.acm.org/10.1145/872757.872838.

[CWL14] Richard Cyganiak, David Wood, and Markus Lanthaler. *RDF 1.1 Concepts and Abstract Syntax*. Tech. rep. W3C, Jan. 2014.

[Das+07] Gautam Das et al. "Ad-hoc Top-k Query Answering for Data Streams". In: *Proceedings of the 33rd International Conference on Very Large Data Bases*. VLDB '07. Vienna, Austria: VLDB Endowment, 2007, pp. 183–194. ISBN: 978-1-59593-649-3. URL: http://dl.acm.org/citation.cfm?id=1325851.1325875.

[Del+14] Daniele Dell'Aglio et al. "RSP-QL Semantics: A Unifying Query Model to Explain Heterogeneity of RDF Stream Processing Systems". In: *Int. J. Semant. Web Inf. Syst.* 10.4 (Oct. 2014), pp. 17–44. ISSN: 1552-6283. DOI: 10.4018/ijswis.2014100102. URL: http://dx.doi.org/10.4018/ijswis.2014100102.

[DFT11] Nihal Dindar, Peter M. Fischer, and Nesime Tatbul. "DejaVu: A Complex Event Processing System for Pattern Matching over Live and Historical Data Streams". In: *Proceedings of the 5th ACM International Conference on Distributed Event-based System*. DEBS '11. New York, New York, USA: ACM, 2011, pp. 399–400. ISBN: 978-1-4503-0423-8. DOI: 10.1145/2002259.2002330. URL: http://doi.acm.org/10.1145/2002259.2002330.

[DGR03] Abhinandan Das, Johannes Gehrke, and Mirek Riedewald. "Approximate Join Processing over Data Streams". In: *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*. SIGMOD '03. San Diego, California: ACM, 2003, pp. 40–51. ISBN: 1-58113-634-X. DOI: 10.1145/872757.872765. URL: http://doi.acm.org/10.1145/872757.872765.

[Dro] *Drool Fusion*. http://www.drools.org/. Accessed: 2016-06-03.

[Dug+15] Jennie Duggan et al. "The BigDAWG Polystore System". In: *SIGMOD Rec.* 44.2 (Aug. 2015), pp. 11–16. ISSN: 0163-5808. DOI: 10.1145/2814710.2814713. URL: http://doi.acm.org/10.1145/2814710.2814713.

[DV+09] Emanuele Della Valle et al. "A First Step Towards Stream Reasoning". In: *Future Internet – FIS 2008: First Future Internet Symposium, FIS 2008 Vienna, Austria, September 29-30, 2008 Revised Selected Papers*. Ed. by John Domingue, Dieter Fensel, and Paolo Traverso. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 72–81. ISBN: 978-3-642-00985-3. DOI: 10.1007/978-3-642-00985-3_6. URL: http://dx.doi.org/10.1007/978-3-642-00985-3_6.

[Eck+11] Michael Eckert et al. "Reasoning in Event-Based Distributed Systems". In: ed. by Sven Helmer, Alexandra Poulovassilis, and Fatos Xhafa. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011. Chap. A CEP Babelfish: Languages for Complex Event Processing and Querying Surveyed, pp. 47–70. ISBN: 978-3-642-19724-6. DOI: 10.1007/978-3-642-19724-6_3. URL: http://dx.doi.org/10.1007/978-3-642-19724-6_3.

[Erl+15] Orri Erling et al. "The LDBC Social Network Benchmark: Interactive Workload". In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. SIGMOD '15. Melbourne, Victoria, Australia: ACM, 2015, pp. 619–630. ISBN: 978-1-4503-2758-9. DOI: 10.1145/2723372.2742786. URL: http://doi.acm.org/10.1145/2723372.2742786.

[Eug+03] Patrick Th. Eugster et al. "The Many Faces of Publish/Subscribe". In: *ACM Comput. Surv.* 35.2 (June 2003), pp. 114–131. ISSN: 0360-0300. DOI: 10.1145/857076.857078. URL: http://doi.acm.org/10.1145/857076.857078.

[FAR11] Paul Fodor, Darko Anicic, and Sebastian Rudolph. "Results on Out-of-Order Event Processing". In: *Practical Aspects of Declarative Languages: 13th International Symposium, PADL 2011, Austin, TX, USA, January 24-25, 2011. Proceedings*. Ed. by Ricardo Rocha and John Launchbury. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 220–234. ISBN: 978-3-642-18378-2. DOI: 10.1007/978-3-642-18378-2_18. URL: http://dx.doi.org/10.1007/978-3-642-18378-2_18.

[For90] Charles L. Forgy. "Expert Systems". In: ed. by Peter G. Raeth. Los Alamitos, CA, USA: IEEE Computer Society Press, 1990. Chap. Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem, pp. 324–341. ISBN: 0-8186-8904-8. URL: http://dl.acm.org/citation.cfm?id=115710.115736.

[Fra+09] Michael J. Franklin et al. "Continuous Analytics: Rethinking Query Processing in a Network-Effect World". In: *CIDR 2009, Fourth Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 4-7, 2009, Online Proceedings*. 2009. URL: http://www-db.cs.wisc.edu/cidr/cidr2009/Paper_122.pdf.

[Gal+09] Ixent Galpin et al. "Comprehensive Optimization of Declarative Sensor Network Queries". In: *Proceedings of the 21st International Conference on Scientific and Statistical Database Management*. SSDBM 2009. New Orleans, LA, USA: Springer-Verlag, 2009, pp. 339–360. ISBN: 978-3-642-02278-4. DOI: 10.1007/978-3-642-02279-1_26. URL: http://dx.doi.org/10.1007/978-3-642-02279-1_26.

[GD94] S. Gatziu and K. R. Dittrich. "Detecting composite events in active database systems using Petri nets". In: *Research Issues in Data Engineering, 1994. Active Database Systems. Proceedings Fourth International Workshop on*. 1994, pp. 2–9. DOI: 10.1109/RIDE.1994.282859.

[Ge+15] Chang Ge et al. "Indexing Bi-temporal Windows". In: *Proceedings of the 27th International Conference on Scientific and Statistical Database Management*. SSDBM '15. La Jolla, California: ACM, 2015, 19:1–19:12. ISBN: 978-1-4503-3709-0. DOI: 10.1145/2791347.2791373. URL: http://doi.acm.org/10.1145/2791347.2791373.

[GFV96] Stella Gatziu, Hans Fritschi, and Anca Vaduva. *SAMOS an Active Object-Oriented Database System: Manual*. Tech. rep. 1996.

[GGÖ04] Lukasz Golab, Shaveen Garg, and M. Tamer Özsu. "On Indexing Sliding Windows over Online Data Streams". In: *Advances in Database Technology - EDBT 2004: 9th International Conference on Extending Database Technology, Heraklion, Crete, Greece, March 14-18, 2004*. Ed. by Elisa Bertino et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 712–729. ISBN: 978-3-540-24741-8. DOI: 10.1007/978-3-540-24741-8_41. URL: http://dx.doi.org/10.1007/978-3-540-24741-8_41.

[Gha+08] Thanaa M. Ghanem et al. "Supporting Views in Data Stream Management Systems". In: *ACM Trans. Database Syst.* 35.1 (Feb. 2008), 1:1–1:47. ISSN: 0362-5915. DOI: 10.1145/1670243.1670244. URL: http://doi.acm.org/10.1145/1670243.1670244.

[GHV05] Claudio Gutierrez, Carlos Hurtado, and Alejandro Vaisman. "Temporal RDF". In: *The Semantic Web: Research and Applications: Second European Semantic Web Conference, ESWC 2005, Heraklion, Crete, Greece, May 29–June 1, 2005. Proceedings*. Ed. by Asunción Gómez-Pérez and Jérôme Euzenat. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 93–107. ISBN: 978-3-540-31547-6. DOI: 10.1007/11431053_7. URL: http://dx.doi.org/10.1007/11431053_7.

[Gil+01] Anna C. Gilbert et al. "Surfing Wavelets on Streams: One-Pass Summaries for Approximate Aggregate Queries". In: *Proceedings of the 27th International Conference on Very Large Data Bases*. VLDB '01. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2001, pp. 79–88. ISBN: 1-55860-804-4. URL: http://dl.acm.org/citation.cfm?id=645927.672174.

[Gil+15] Syed Gillani et al. "Top-K Queries in RDF Graph-based Stream Processing with Actors". In: *Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems*. DEBS '15. Oslo, Norway: ACM, 2015, pp. 293–300. ISBN: 978-1-4503-3286-6. DOI: 10.1145/2675743.2772587. URL: http://doi.acm.org/10.1145/2675743.2772587.

[GJS92a] N. H. Gehani, H. V. Jagadish, and O. Shmueli. "Event Specification in an Active Object-oriented Database". In: *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data*. SIGMOD '92. San Diego, California, USA: ACM, 1992, pp. 81–90. ISBN: 0-89791-521-6. DOI: 10.1145/130283.130300. URL: http://doi.acm.org/10.1145/130283.130300.

[GJS92b] Narain H. Gehani, H. V. Jagadish, and Oded Shmueli. "Composite Event Specification in Active Databases: Model &Amp; Implementation". In: *Proceedings of the 18th International Conference on Very Large Data Bases*. VLDB '92. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1992, pp. 327–338. ISBN: 1-55860-151-1. URL: http://dl.acm.org/citation.cfm?id=645918.672484.

[GK02] Sudipto Guha and Nick Koudas. "Approximating a Data Stream for Querying and Estimation: Algorithms and Performance Evaluation." In: *ICDE*. Ed. by Rakesh Agrawal and Klaus R. Dittrich. IEEE Computer Society, 2002, pp. 567–576. ISBN: 0-7695-1531-2. URL: http://dblp.uni-trier.de/db/conf/icde/icde2002.html#GuhaK02.

[GLP+14] Syed Gillani, Frederique Laforest, Gauthier Picard, et al. "A Generic Ontology for Prosumer-Oriented Smart Grid." In: *EDBT/ICDT Workshops*. 2014, pp. 134–139.

[GLP14] Syed Gillani, Frédérique Laforest, and Gauthier Picard. "A Generic Ontology for Prosumer-Oriented Smart Grid". In: *Proceedings of the Workshops of the EDBT/ICDT 2014 Joint Conference (EDBT/ICDT 2014), Athens, Greece, March 28, 2014*. 2014, pp. 134–139. URL: http://ceur-ws.org/Vol-1133/paper-21.pdf.

[GM06] Sudipto Guha and Andrew McGregor. "Approximate Quantiles and the Order of the Stream". In: *Proceedings of the Twenty-fifth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*. PODS '06. Chicago, IL, USA: ACM, 2006, pp. 273–279. ISBN: 1-59593-318-2. DOI: 10.1145/1142351.1142390. URL: http://doi.acm.org/10.1145/1142351.1142390.

[GO03] Lukasz Golab and M. Tamer Özsu. "Issues in Data Stream Management". In: *SIGMOD Rec.* 32.2 (June 2003), pp. 5–14. ISSN: 0163-5808. DOI: 10.1145/776985.776986. URL: http://doi.acm.org/10.1145/776985.776986.

[GPH05] Yuanbo Guo, Zhengxiang Pan, and Jeff Heflin. "LUBM: A Benchmark for OWL Knowledge Base Systems". In: *Web Semant.* 3.2-3 (Oct. 2005), pp. 158–182. ISSN: 1570-8268. DOI: 10.1016/j.websem.2005.06.005. URL: http://dx.doi.org/10.1016/j.websem.2005.06.005.

[GPL14] Syed Gillani, Gauthier Picard, and Frédérique Laforest. "IntelSCEP: Towards an Intelligent Semantic Complex Event Processing Framework for Prosumer-Oriented SmartGrid". In: *Proceedings of the 2014 International Workshop on Web Intelligence and Smart Sensing*. IWWISS '14. Saint Etienne, France: ACM, 2014, 23:1–23:2. ISBN: 978-1-4503-2747-3. DOI: 10.1145/2637064.2637110. URL: http://doi.acm.org/10.1145/2637064.2637110.

[GPL16a] Syed Gillani, Gauthier Picard, and Frédérique Laforest. "Continuous Graph Pattern Matching over Knowledge Graph Streams". In: *Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems*. DEBS '16. Irvine, California: ACM, 2016, pp. 214–225. ISBN: 978-1-4503-4021-2. DOI: 10.1145/2933267.2933306. URL: http://doi.acm.org/10.1145/2933267.2933306.

[GPL16b] Syed Gillani, Gauthier Picard, and Frédérique Laforest. "DIONYSUS: Towards Query-aware Distributed Processing of RDF Graph Streams". In: *Proceedings of the Workshops of the EDBT/ICDT 2016 Joint Conference, EDBT/ICDT Workshops 2016, Bordeaux, France, March 15, 2016*. 2016. URL: http://ceur-ws.org/Vol-1558/paper22.pdf.

[GPL16c] Syed Gillani, Gauthier Picard, and Frédérique Laforest. "SPECTRA: Continuous Query Processing for RDF Graph Streams Over Sliding Windows". In: *Proceedings of the 28th International Conference on Scientific and Statistical Database Management*. SSDBM '16. Budapest, Hungary: ACM, 2016, 17:1–17:12. ISBN: 978-1-4503-4215-5. DOI: 10.1145/2949689.2949701. URL: http://doi.acm.org/10.1145/2949689.2949701.

[Gra10] Fabio Grandi. "T-SPARQL: a TSQL2-like temporal query language for RDF". In: *In International Workshop on on Querying Graph Structured Data*. 2010, pp. 21–30.

[Gur+14] Sairam Gurajada et al. "TriAD: A Distributed Shared-nothing RDF Engine Based on Asynchronous Message Passing". In: *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*. SIGMOD '14. Snowbird, Utah, USA: ACM, 2014, pp. 289–300. ISBN: 978-1-4503-2376-5. DOI: 10.1145/2588555.2610511. URL: http://doi.acm.org/10.1145/2588555.2610511.

[Ham+03] Moustafa Hammad et al. *Efficient Execution of Sliding-Window Queries Over Data Streams*. 2003.

[HD05] Andreas Harth and Stefan Decker. "Optimized Index Structures for Querying RDF from the Web." In: *LA-WEB*. IEEE Computer Society, 2005, pp. 71–80. ISBN: 0-7695-2471-0. URL: http://dblp.uni-trier.de/db/conf/la-web/la-web2005.html#HarthD05.

[Hei+14] Thomas Heinze et al. "Latency-aware Elastic Scaling for Distributed Data Stream Processing Systems". In: *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems*. DEBS '14. Mumbai, India: ACM, 2014, pp. 13–22. ISBN: 978-1-4503-2737-4. DOI: 10.1145/2611286.2611294. URL: http://doi.acm.org/10.1145/2611286.2611294.

[Hen+09] M. Hentschel et al. "Scalable Data Integration by Mapping Data to Queries". In: *month* 7.633 (2009), p. 26.

[HG04] Jonathan Hayes and Claudio Gutierrez. "Bipartite Graphs as Intermediate Model for RDF". In: *The Semantic Web – ISWC 2004: Third International Semantic Web Conference, Hiroshima, Japan, November 7-11, 2004. Proceedings*. Ed. by Sheila A. McIlraith, Dimitris Plexousakis, and Frank van Harmelen. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 47–61. ISBN: 978-3-540-30475-3. DOI: 10.1007/978-3-540-30475-3_5. URL: http://dx.doi.org/10.1007/978-3-540-30475-3_5.

[Hog+14] Aidan Hogan et al. "Everything you always wanted to know about blank nodes". In: *Web Semantics: Science, Services and Agents on the World Wide Web* 27–28 (2014). Semantic Web Challenge 2013, pp. 42 –69. ISSN: 1570-8268. DOI: http://dx.doi.org/10.1016/j.websem.2014.06.004. URL: http://www.sciencedirect.com/science/article/pii/S1570826814000481.

[HV02] A. Hinze and A. Voisard. "A parameterized algebra for event notification services". In: *Temporal Representation and Reasoning, 2002. TIME 2002. Proceedings.Ninth International Symposium on*. 2002, pp. 61–63. DOI: 10.1109/TIME.2002.1027476.

[HV05] Jayant R. Haritsa and T. M. Vijayaraman, eds. *Advances in Data Management 2005, Proceedings of the Eleventh International Conference on Management of Data, January 6, 7, and 8, 2005, Goa, India*. Computer Society of India, 2005.

[Idr+11] Stratos Idreos et al. "Merging What's Cracked, Cracking What's Merged: Adaptive Indexing in Main-memory Column-stores". In: *Proc. VLDB Endow.* 4.9 (June 2011), pp. 586–597. ISSN: 2150-8097. DOI: 10.14778/2002938.2002944. URL: http://dx.doi.org/10.14778/2002938.2002944.

[IKM07] Stratos Idreos, Martin L. Kersten, and Stefan Manegold. "Updating a Cracked Database". In: *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*. SIGMOD '07. Beijing, China: ACM, 2007, pp. 413–424. ISBN: 978-1-59593-686-8. DOI: 10.1145/1247480.1247527. URL: http://doi.acm.org/10.1145/1247480.1247527.

[Jen94] Kurt Jensen. "An Introduction to the Theoretical Aspects of Coloured Petri Nets". In: *A Decade of Concurrency, Reflections and Perspectives, REX School/Symposium*. London, UK, UK: Springer-Verlag, 1994, pp. 230–272. ISBN: 3-540-58043-3. URL: http://dl.acm.org/citation.cfm?id=648145.750149.

[Kal+08] Robert Kallman et al. "H-store: A High-performance, Distributed Main Memory Transaction Processing System". In: *Proc. VLDB Endow.* 1.2 (Aug. 2008), pp. 1496–1499. ISSN: 2150-8097. DOI: 10.14778/1454159.1454211. URL: http://dx.doi.org/10.14778/1454159.1454211.

[Kam+16] Abderrahmen Kammoun et al. "High Performance Top-k Processing of Non-linear Windows over Data Streams". In: *Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems*. DEBS '16. Irvine, California: ACM, 2016, pp. 293–300. ISBN: 978-1-4503-4021-2. DOI: 10.1145/2933267.2933507. URL: http://doi.acm.org/10.1145/2933267.2933507.

[KCF12] Srdjan Komazec, Davide Cerri, and Dieter Fensel. "Sparkwave: Continuous Schema-enhanced Pattern Matching over RDF Data Streams". In: *Proceedings of the 6th ACM International Conference on Distributed Event-Based Systems*. DEBS '12. Berlin, Germany: ACM, 2012, pp. 58–68. ISBN: 978-1-4503-1315-5. DOI: 10.1145/2335484.2335491. URL: http://doi.acm.org/10.1145/2335484.2335491.

[KKM13] Bruce M. Kapron, Valerie King, and Ben Mountjoy. "Dynamic graph connectivity in polylogarithmic worst case time." In: *SODA*. 2013, pp. 1131–1142. URL: http://dblp.uni-trier.de/db/conf/soda/soda2013.html#KapronKM13.

[KNV04] Jaewoo Kang, Jeffrey F. Naughton, and Stratis Viglas. "Evaluating Window Joins over Unbounded Streams." In: *ICDE*. Ed. by Umeshwar Dayal, Krithi Ramamritham, and T. M. Vijayaraman. IEEE Computer Society, Mar. 31, 2004, pp. 341–352. ISBN: 0-7803-7665-X. URL: http://dblp.uni-trier.de/db/conf/icde/icde2003.html#KangNV03.

[Koz+06] Alex Kozlenkov et al. "Current Trends in Database Technology – EDBT 2006: EDBT 2006 Workshops PhD, DataX, IIDB, IIHA, ICSNW, QLQP, PIM, PaRMA, and Reactivity on the Web, Munich, Germany, March 26-31, 2006, Revised Selected Papers". In: ed. by Torsten Grust et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006. Chap. Prova: Rule-Based Java Scripting for Distributed Web Applications: A Case Study in Bioinformatics, pp. 899–908. ISBN: 978-3-540-46790-8. DOI: 10.1007/11896548_68. URL: http://dx.doi.org/10.1007/11896548_68.

[KS09a] Jürgen Krämer and Bernhard Seeger. "Semantics and Implementation of Continuous Sliding Window Queries over Data Streams". In: *ACM Trans. Database Syst.* 34.1 (Apr. 2009), 4:1–4:49. ISSN: 0362-5915. DOI: 10.1145/1508857.1508861. URL: http://doi.acm.org/10.1145/1508857.1508861.

[KS09b] Jürgen Krämer and Bernhard Seeger. "Semantics and Implementation of Continuous Sliding Window Queries over Data Streams". In: *ACM Trans. Database Syst.* Vol. 34. 2009, 4:1–4:49.

[KS86] R Kowalski and M Sergot. "A Logic-based Calculus of Events". In: *New Gen. Comput.* 4.1 (Jan. 1986), pp. 67–95. ISSN: 0288-3635. DOI: 10.1007/BF03037383. URL: http://dx.doi.org/10.1007/BF03037383.

[KS92] Michael Kifer and V.S. Subrahmanian. "Theory of generalized annotated logic programming and its applications**A preliminary report on this research has appeared in [34]." In: *The Journal of Logic Programming* 12.4 (1992), pp. 335 –367. ISSN: 0743-1066. DOI: http://dx.doi.org/10.1016/0743-1066(92)90007-P. URL: http://www.sciencedirect.com/science/article/pii/074310669290007P.

[LB15] Feilong Liu and Spyros Blanas. "Forecasting the Cost of Processing Multi-join Queries via Hashing for Main-memory Databases". In: *soCC*. 2015, pp. 153–166.

[Let+10] J. Letchner et al. "Approximation trade-offs in Markovian stream processing: An empirical study". In: *2010 IEEE 26th International Conference on Data Engineering (ICDE 2010)*. 2010, pp. 936–939. DOI: 10.1109/ICDE.2010.5447926.

[LGI09] Erietta Liarou, Romulo Goncalves, and Stratos Idreos. "Exploiting the Power of Relational Databases for Efficient Stream Processing". In: *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology*. EDBT '09. Saint Petersburg, Russia: ACM, 2009, pp. 323–334. ISBN: 978-1-60558-422-5. DOI: 10.1145/1516360.1516398. URL: http://doi.acm.org/10.1145/1516360.1516398.

[Li+07] M. Li et al. "Event Stream Processing with Out-of-Order Data Arrival". In: *Distributed Computing Systems Workshops, 2007. ICDCSW '07. 27th International Conference on*. 2007, pp. 67–67. DOI: 10.1109/ICDCSW.2007.35.

[Li+08] Jin Li et al. "Out-of-order Processing: A New Architecture for High-performance Stream Systems". In: *Proc. VLDB Endow.* 1.1 (Aug. 2008), pp. 274–288. ISSN: 2150-8097. DOI: 10.14778/1453856.1453890. URL: http://dx.doi.org/10.14778/1453856.1453890.

[LLM98] Georg Lausen, Bertram Ludäscher, and Wolfgang May. "Transactions and Change in Logic Databases: International Seminar on Logic Databases and the Meaning of Change Schloss Dagstuhl, Germany, September 23–27, 1996 and ILPS'97 Post-Conference Workshop on (Trans)Actions and Change in Logic Programming and Deductive Databases, (DYNAMICS'97) Port Jefferson, NY, USA, October 17, 1997 Invited Surveys and Selected Papers". In: ed. by Burkhard Freitag et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998. Chap. On active deductive databases: The statelog approach, pp. 69–106. ISBN: 978-3-540-49449-2. DOI: 10.1007/BFb0055496. URL: http://dx.doi.org/10.1007/BFb0055496.

[LP+11] Danh Le-Phuoc et al. "A Native and Adaptive Approach for Unified Processing of Linked Streams and Linked Data". In: *Proceedings of the 10th International Conference on The Semantic Web - Volume Part I*. ISWC'11. Bonn, Germany: Springer-Verlag, 2011, pp. 370–388. ISBN: 978-3-642-25072-9. URL: http://dl.acm.org/citation.cfm?id=2063016.2063041.

[Luc01] David C. Luckham. *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2001. ISBN: 0201727897.

[LWZ04] Yan-Nei Law, Haixun Wang, and Carlo Zaniolo. "Query Languages and Data Models for Database Sequences and Data Streams". In: *Very Large Data Bases (VLDB)*. 2004, pp. 492–503. URL: http://www.vldb.org/conf/2004/RS12P2.PDF.

[MC13] Jose Mora and Oscar Corcho. "Engineering Optimisations in Query Rewriting for OBDA". In: *Proceedings of the 9th International Conference on Semantic Systems*. I-SEMANTICS '13. Graz, Austria: ACM, 2013, pp. 41–48. ISBN: 978-1-4503-1972-0. DOI: 10.1145/2506182.2506188. URL: http://doi.acm.org/10.1145/2506182.2506188.

[ME01] D. Moreto and M. Endler. "Evaluating composite events using shared trees". In: *IEE Proceedings - Software* 148.1 (2001), pp. 1–10. ISSN: 1462-5970. DOI: 10.1049/ip-sen:20010241.

[Mee+15] John Meehan et al. "S-Store: Streaming Meets Transaction Processing". In: *Proc. VLDB Endow.* 8.13 (Sept. 2015), pp. 2134–2145. ISSN: 2150-8097. DOI: 10.14778/2831360.2831367. URL: http://dx.doi.org/10.14778/2831360.2831367.

[MH04] Deborah L. McGuinness and Frank van Harmelen. *OWL Web Ontology Language Overview*. Tech. rep. REC-owl-features-20040210. W3C, 2004.

[MM09] Yuan Mei and Samuel Madden. "ZStream: A Cost-based Query Processor for Adaptively Detecting Composite Events". In: *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*. SIGMOD '09. Providence, Rhode Island, USA: ACM, 2009, pp. 193–206. ISBN: 978-1-60558-551-2. DOI: 10.1145/1559845.1559867. URL: http://doi.acm.org/10.1145/1559845.1559867.

[Moz+13] Barzan Mozafari et al. "High-performance Complex Event Processing over Hierarchical Data". In: *ACM Trans. Database Syst.* 38.4 (Dec. 2013), 21:1–21:39. ISSN: 0362-5915. DOI: 10.1145/2536779. URL: http://doi.acm.org/10.1145/2536779.

[MPG09] Sergio Muñoz, Jorge Pérez, and Claudio Gutierrez. "Simple and Efficient Minimal {RDFS}". In: *Web Semantics: Science, Services and Agents on the World Wide Web* 7.3 (2009). The Web of Data, pp. 220 –234. ISSN: 1570-8268. DOI: http://dx.doi.org/10.1016/j.websem.2009.07.003. URL: http://www.sciencedirect.com/science/article/pii/S1570826809000249.

[MSS97] M. Mansouri-Samani and M. Sloman. "GEM: A generalized event monitoring language for distributed systems". In: *Distributed Systems Engineering* 4 (1997), pp. 96–108.

[MZ95] Iakovos Motakis and Carlo Zaniolo. "Recent Advances in Temporal Databases: Proceedings of the International Workshop on Temporal Databases, Zurich, Switzerland, 17–18 September 1995". In: ed. by James Clifford and Alexander Tuzhilin. London: Springer London, 1995. Chap. Composite Temporal Events in Active Databases: A Formal Semantics, pp. 332–351. ISBN: 978-1-4471-3033-8. DOI: 10.1007/978-1-4471-3033-8_18. URL: http://dx.doi.org/10.1007/978-1-4471-3033-8_18.

[NCT08] Alex Russakovsky Neil Conway Michael J. Franklin and Neil Thombre. "TruSQL: A Stream-Relational Extension to SQL". In: *Technical Report, Truviso, Inc.* 2008.

[Nel65] T. H. Nelson. "Complex Information Processing: A File Structure for the Complex, the Changing and the Indeterminate". In: *Proceedings of the 1965 20th National Conference*. ACM '65. Cleveland, Ohio, USA: ACM, 1965, pp. 84–100. DOI: 10.1145/800197.806036. URL: http://doi.acm.org/10.1145/800197.806036.

[Nen+15] Yavor Nenov et al. "RDFox: A Highly-Scalable RDF Store." In: *International Semantic Web Conference (ISWC)*. Vol. 9367. Lecture Notes in Computer Science. Springer, 2015, pp. 3–20. ISBN: 978-3-319-25009-0. URL: http://dblp.uni-trier.de/db/conf/semweb/iswc2015-2.html#NenovPMHWB15.

[Neu+10] Leonardo Neumeyer et al. "S4: Distributed Stream Computing Platform". In: *Proceedings of the 2010 IEEE International Conference on Data Mining Workshops*. ICDMW '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 170–177. ISBN: 978-0-7695-4257-7. DOI: 10.1109/ICDMW.2010.172. URL: http://dx.doi.org/10.1109/ICDMW.2010.172.

[NW10a] Thomas Neumann and Gerhard Weikum. "The RDF-3X Engine for Scalable Management of RDF Data". In: *VLDB*. 2010, pp. 91–113. DOI: 10.1007/s00778-009-0165-y. URL: http://dx.doi.org/10.1007/s00778-009-0165-y.

[NW10b] Thomas Neumann and Gerhard Weikum. "The RDF-3X Engine for Scalable Management of RDF Data". In: *The VLDB Journal* 19.1 (Feb. 2010), pp. 91–113. ISSN: 1066-8888. DOI: 10.1007/s00778-009-0165-y. URL: http://dx.doi.org/10.1007/s00778-009-0165-y.

[PAG09a] Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez. "Semantics and Complexity of SPARQL". In: *ACM Trans. Database Syst.* 34.3 (Sept. 2009), 16:1–16:45. ISSN: 0362-5915. DOI: 10.1145/1567274.1567278. URL: http://doi.acm.org/10.1145/1567274.1567278.

[PAG09b] Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez. "Semantics and complexity of SPARQL". In: *ACM Transactions on Database Systems*. Vol. 34. 2009, pp. 1–45.

[Pas06] Adrian Paschke. "ECA-LP / ECA-RuleML: A Homogeneous Event-Condition-Action Logic Programming Language". In: *CoRR* abs/cs/0609143 (2006). URL: http://arxiv.org/abs/cs/0609143.

[PD99] Norman W. Paton and Oscar Díaz. "Active Database Systems". In: *ACM Comput. Surv.* 31.1 (Mar. 1999), pp. 63–103. ISSN: 0360-0300. DOI: 10.1145/311531.311623. URL: http://doi.acm.org/10.1145/311531.311623.

[Pic+12] François Picalausa et al. "A Structural Approach to Indexing Triples". In: *The Semantic Web: Research and Applications: 9th Extended Semantic Web Conference, ESWC 2012, Heraklion, Crete, Greece, May 27-31, 2012. Proceedings*. Ed. by Elena Simperl et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 406–421. ISBN: 978-3-642-30284-8. DOI: 10.1007/978-3-642-30284-8_34. URL: http://dx.doi.org/10.1007/978-3-642-30284-8_34.

[PS08] Eric Prud'hommeaux and Andy Seaborne. *SPARQL Query Language for RDF*. Ed. by W3C Recommendation. Latest version available as http://www.w3.org/TR/rdf-sparql-query/. 2008. URL: http://www.w3.org/TR/2008/REC-rdf-sparql-query-20080115/.

[PUHM09] Héctor Pérez-Urbina, Ian Horrocks, and Boris Motik. "Efficient Query Answering for OWL 2". In: *The Semantic Web - ISWC 2009: 8th International Semantic Web Conference, ISWC 2009, Chantilly, VA, USA, October 25-29, 2009. Proceedings*. Ed. by Abraham Bernstein et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 489–504. ISBN: 978-3-642-04930-9. DOI: 10.1007/978-3-642-04930-9_31. URL: http://dx.doi.org/10.1007/978-3-642-04930-9_31.

[Rdf] "Resource Description Framework (RDF) Model and Syntax Specification. W3C Recommendation. 1999". In: *W3C*. 1999. URL: http://www.w3.org/TR/1999/REC-rdf-syntax-19990222/.

[RNT12] Mikko Rinne, Esko Nuutila, and Seppo Törmä. "INSTANS: High-Performance Event Processing with Standard RDF and SPARQL". In: *Proceedings of the ISWC 2012 Posters & Demonstrations Track, Boston, USA, November 11-15, 2012*. 2012. URL: http://ceur-ws.org/Vol-914/paper_22.pdf.

[Ron98] Claudia L. Roncancio. "Active, Real-Time, and Temporal Database Systems: Second International Workshop, ARTDB-97 Como, Italy, September 8–9, 1997 Proceedings". In: ed. by Sten F. Andler and Jörgen Hansson. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998. Chap. Toward Duration-Based, Constrained and Dynamic Event Types, pp. 176–193. ISBN: 978-3-540-49151-4. DOI: 10.1007/3-540-49151-1_10. URL: http://dx.doi.org/10.1007/3-540-49151-1_10.

[SB05] Marco Seiriö and Mikael Berndtsson. "Design and Implementation of an ECA Rule Markup Language". In: *Proceedings of the First International Conference on Rules and Rule Markup Languages for the Semantic Web*. RuleML'05. Galway, Ireland: Springer-Verlag, 2005, pp. 98–112. ISBN: 3-540-29922-X, 978-3-540-29922-6. DOI: 10.1007/11580072_9. URL: http://dx.doi.org/10.1007/11580072_9.

[Sch+06] Karl Schnaitter et al. "COLT: Continuous On-line Tuning". In: *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*. SIGMOD '06. Chicago, IL, USA: ACM, 2006, pp. 793–795. ISBN: 1-59593-434-0. DOI: 10.1145/1142473.1142592. URL: http://doi.acm.org/10.1145/1142473.1142592.

[ScZ05] Michael Stonebraker, Uğur Çetintemel, and Stan Zdonik. "The 8 Requirements of Real-time Stream Processing". In: *SIGMOD Rec.* 34.4 (Dec. 2005), pp. 42–47. ISSN: 0163-5808. DOI: 10.1145/1107499.1107504. URL: http://doi.acm.org/10.1145/1107499.1107504.

[SG07] Sharmila Subramaniam and Dimitrios Gunopulos. "Data Streams: Models and Algorithms". In: ed. by Charu C. Aggarwal. Boston, MA: Springer US, 2007. Chap. A Survey of Stream Processing Problems and Techniques in Sensor Networks, pp. 333–352. ISBN: 978-0-387-47534-9. DOI: 10.1007/978-0-387-47534-9_15. URL: http://dx.doi.org/10.1007/978-0-387-47534-9_15.

[SGL14] Gauthier Picard Syed Gillani and Frédérique Laforest. "Towards Efficient Semantically Enriched Complex Event Processing and Pattern Matching". In: *Proceedings of the 3rd International Workshop on Ordering and Reasoning Co-located with the 13th International Semantic Web Conference (ISWC 2014), Riva del Garda, Italy, October 20th, 2014*. 2014, pp. 47–54. URL: http://ceur-ws.org/Vol-1303/paper_2.pdf.

[SH13] Andy Seaborne Steve Harris Garlik. *SPARQL 1.1 Query Language*. Ed. by W3C Recommendation. Latest version available as https://www.w3.org/TR/2013/REC-sparql11-query-20130321/. 2013.

[Sim+16] Natcha Simsiri et al. "Work-Efficient Parallel and Incremental Graph Connectivity". In: *CoRR*. Vol. abs/1602.05232. 2016.

[SMMP09] Nicholas Poul Schultz-Møller, Matteo Migliavacca, and Peter Pietzuch. "Distributed Complex Event Processing with Query Rewriting". In: *Proceedings of the Third ACM International Conference on Distributed Event-Based Systems*. DEBS '09. Nashville, Tennessee: ACM, 2009, 4:1–4:12. ISBN: 978-1-60558-665-6. DOI: 10.1145/1619258.1619264. URL: http://doi.acm.org/10.1145/1619258.1619264.

[SSS08] Kay-Uwe Schmidt, Roland Stühmer, and Ljiljana Stojanovic. "Blending Complex Event Processing with the RETE Algorithm". In: *iCEP2008: 1st International workshop on Complex Event Processing for the Future Internet colocated with the Future Internet Symposium (FIS2008)*. Ed. by Darko Anicic et al. Vol. Vol-412. CEUR Workshop Proceedings (CEUR-WS.org, ISSN 1613-0073), 2008. URL: http://ceur-ws.org/Vol-412/paper3.pdf.

[Sub+16] Julien Subercaze et al. "Inferray: Fast In-memory RDF Inference". In: *Proc. VLDB Endow.* 9.6 (Jan. 2016), pp. 468–479. ISSN: 2150-8097. DOI: 10.14778/2904121.2904123. URL: http://dx.doi.org/10.14778/2904121.2904123.

[Sán+05] César Sánchez et al. "Formal Techniques for Networked and Distributed Systems - FORTE 2005: 25th IFIP WG 6.1 International Conference, Taipei, Taiwan, October 2-5, 2005. Proceedings". In: ed. by Farn Wang. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005. Chap. Expressive Completeness of an Event-Pattern Reactive Programming Language, pp. 529–532. ISBN: 978-3-540-32084-5. DOI: 10.1007/11562436_39. URL: http://dx.doi.org/10.1007/11562436_39.

[Tan+15] Kanat Tangwongsan et al. "General Incremental Sliding-window Aggregation". In: *Proc. VLDB Endow.* 8.7 (Feb. 2015), pp. 702–713. ISSN: 2150-8097. DOI: 10.14778/2752939.2752940. URL: http://dx.doi.org/10.14778/2752939.2752940.

[Tat+03] Nesime Tatbul et al. "Load Shedding in a Data Stream Manager". In: *Proceedings of the 29th International Conference on Very Large Data Bases - Volume 29*. VLDB '03. Berlin, Germany: VLDB Endowment, 2003, pp. 309–320. ISBN: 0-12-722442-4. URL: http://dl.acm.org/citation.cfm?id=1315451.1315479.

[Tat10] N. Tatbul. "Streaming data integration: Challenges and opportunities". In: *Data Engineering Workshops (ICDEW), 2010 IEEE 26th International Conference on*. 2010, pp. 155–158. DOI: 10.1109/ICDEW.2010.5452751.

[TB09] Jonas Tappolet and Abraham Bernstein. "Applied Temporal RDF: Efficient Temporal Querying of RDF Data with SPARQL". In: *Proceedings of the 6th European Semantic Web Conference on The Semantic Web: Research and Applications*. ESWC 2009 Heraklion. Heraklion, Crete, Greece: Springer-Verlag, 2009, pp. 308–322. ISBN: 978-3-642-02120-6. DOI: 10.1007/978-3-642-02121-3_25. URL: http://dx.doi.org/10.1007/978-3-642-02121-3_25.

[Tho68] Ken Thompson. "Programming Techniques: Regular Expression Search Algorithm". In: *Commun. ACM* 11.6 (June 1968), pp. 419–422. ISSN: 0001-0782. DOI: 10.1145/363347.363387. URL: http://doi.acm.org/10.1145/363347.363387.

[Tib] *TIBCO Business Events*. http://www.tibco.com/products/event-processing/complex-event-processing/businessevents/. Accessed: 2016-06-03.

[Tuc+03] P. A. Tucker et al. "Exploiting punctuation semantics in continuous data streams". In: *IEEE Transactions on Knowledge and Data Engineering* 15.3 (2003), pp. 555–568. ISSN: 1041-4347. DOI: 10.1109/TKDE.2003.1198390.

[URS10] Octavian Udrea, Diego Reforgiato Recupero, and V. S. Subrahmanian. "Annotated RDF". In: *ACM Trans. Comput. Logic* 11.2 (Jan. 2010), 10:1–10:41. ISSN: 1529-3785. DOI: 10.1145/1656242.1656245. URL: http://doi.acm.org/10.1145/1656242.1656245.

[WDR06a] Eugene Wu, Yanlei Diao, and Shariq Rizvi. "High-performance Complex Event Processing over Streams". In: *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*. SIGMOD '06. Chicago, IL, USA: ACM, 2006, pp. 407–418. ISBN: 1-59593-434-0. DOI: 10.1145/1142473.1142520. URL: http://doi.acm.org/10.1145/1142473.1142520.

[WDR06b] Eugene Wu, Yanlei Diao, and Shariq Rizvi. "High-performance Complex Event Processing over Streams". In: *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*. SIGMOD '06. Chicago, IL, USA: ACM, 2006, pp. 407–418. ISBN: 1-59593-434-0. DOI: 10.1145/1142473.1142520. URL: http://doi.acm.org/10.1145/1142473.1142520.

[WKB08] Cathrin Weiss, Panagiotis Karras, and Abraham Bernstein. "Hexastore: Sextuple Indexing for Semantic Web Data Management". In: *Proc. VLDB Endow.* 1.1 (Aug. 2008), pp. 1008–1019. ISSN: 2150-8097. DOI: 10.14778/1453856.1453965. URL: http://dx.doi.org/10.14778/1453856.1453965.

[WLC14] David Wood, Markus Lanthaler, and Richard Cyganiak. "RDF 1.1 Concepts and Abstract Syntax". In: *W3C Recommendation, Technical Report*. 2014. URL: http://www.w3.org/TR/2014/REC-rdf11-concepts-20140225/ (visited on 03/15/2015).

[YG02] Yong Yao and Johannes Gehrke. "The Cougar Approach to In-network Query Processing in Sensor Networks". In: *SIGMOD Rec.* 31.3 (Sept. 2002), pp. 9–18. ISSN: 0163-5808. DOI: 10.1145/601858.601861. URL: http://doi.acm.org/10.1145/601858.601861.

[ZDI10] Haopeng Zhang, Yanlei Diao, and Neil Immerman. "Recognizing Patterns in Streams with Imprecise Timestamps". In: *Proc. VLDB Endow.* 3.1-2 (Sept. 2010), pp. 244–255. ISSN: 2150-8097. DOI: 10.14778/1920841.1920875. URL: http://dx.doi.org/10.14778/1920841.1920875.

[ZDI14] Haopeng Zhang, Yanlei Diao, and Neil Immerman. "On Complexity and Optimization of Expensive Queries in Complex Event Processing". In: *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*. SIGMOD '14. Snowbird, Utah, USA: ACM, 2014, pp. 217–228. ISBN: 978-1-4503-2376-5. DOI: 10.1145/2588555.2593671. URL: http://doi.acm.org/10.1145/2588555.2593671.

[Zou+11] Lei Zou et al. “gStore: Answering SPARQL Queries via Subgraph Matching”. In: *Proc. VLDB Endow.* 4.8 (May 2011), pp. 482–493. ISSN: 2150-8097. DOI: 10.14778/2002974.2002976. URL: http://dx.doi.org/10.14778/2002974.2002976.

[ZS01] Dong Zhu and A. S. Sethi. “SEL, a new event pattern specification language for event correlation”. In: *Computer Communications and Networks, 2001. Proceedings. Tenth International Conference on*. 2001, pp. 586–589. DOI: 10.1109/ICCCN.2001.956327.

[ÖMN14] Özgür Lütfü Özçep, Ralf Möller, and Christian Neuenstadt. “KI 2014: Advances in Artificial Intelligence: 37th Annual German Conference on AI, Stuttgart, Germany, September 22-26, 2014. Proceedings”. In: ed. by Carsten Lutz and Michael Thielscher. Cham: Springer International Publishing, 2014. Chap. A Stream-Temporal Query Language for Ontology Based Data Access, pp. 183–194. ISBN: 978-3-319-11206-0. DOI: 10.1007/978-3-319-11206-0_18. URL: http://dx.doi.org/10.1007/978-3-319-11206-0_18.