Semantic Web 1 (2017) 1–40
IOS Press

A Query Language for Semantic Complex Event Processing: Syntax, Semantics and Implementation

Editor(s): Name Surname, University, Country
Solicited review(s): Name Surname, University, Country
Open review(s): Name Surname, University, Country

Syed Gillani [a], Antoine Zimmermann [b], Gauthier Picard [b] and Frédérique Laforest [a]
[a] *Univ Lyon, UJM Saint-Étienne, CNRS, Laboratoire Hubert Curien UMR 5516, France*
E-mail: syed.gillani@univ-st-etienne.com, frederique.laforest@telecom-st-etienne.fr
[b] *Univ Lyon, MINES Saint-Étienne, CNRS, Laboratoire Hubert Curien UMR 5516, France*
E-mail: antoine.zimmermann@emse.fr, gauthier.picard@emse.fr

Abstract

Today most applications on the Web and in enterprises produce data in a continuous manner under the form of streams, which have to be handled by Data Stream Management Systems (DSMSs) and Complex Event Processing (CEP) systems. The Semantics Web, through its standards and technologies, is in constant pursue to provide solutions for such paradigms while employing the RDF data model. The integration of Semantic Web technologies in this context can handle the heterogeneity, integration and interpretation of data streams at semantic level. In this paper, we propose a new query language, called SPAseq, that extends SPARQL with new Semantic Complex Event Processing (SCEP) operators that can be evaluated over RDF graph-based events. The novelties of SPAseq includes (i) the expressibility of temporal operators such as *Kleene+*, conjunction, disjunction and event selection strategies, and (ii) the support for multiple heterogeneous streams. SPAseq not only enjoys good expressiveness but also uses a non-deterministic automata (NFA) model for an efficient evaluation of the SPAseq queries. We provide the syntax and semantics of SPAseq and based on this, we implement a query engine that employs NFA to evaluate these operators in an optimised manner. Moreover, we also present an experimental evaluation of its performance, showing that it improves over state-of-the-art approaches.

Keywords: Complex Event Processing, Query Language, Semantic Web, SPARQL, RDF streams, Automata Model, Query Optimisation

1. Introduction

With the evolution of social networks and sensor networks, large volumes of data are generated in a streaming fashion. This leads to the popularity of stream processing systems, where query operators such as selection, aggregation, filtering of data are performed in real-time manner [1,2]. Complex Event Processing (CEP) systems, however, provide a different view and additional operators for these applications: each data item within data streams is considered as an event and predefined temporal patterns are used to generate actions to the systems, people and devices. CEP systems have demonstrated utility in a variety of applications including financial trading, security monitoring, social and sensor network analysis [3,4,5]. In general, CEP denotes algorithmic methods for making sense of events by deriving higher-level knowledge, or complex events, from lower-level events in a timely fashion. CEP applications commonly involve three requirements:

(i) complex predicates (filtering, correlation);
(ii) temporal, order and sequential patterns; and
(iii) transforming event(s) into more complex structures [6,7].

The past several years have seen a large number of CEP systems and query languages developed by both academic and industrial world [5,8,9,10,11,12]. However, most of the existing CEP systems consider a relational data model for streams and their proposed languages and optimisations are also tightly coupled with the model. Hence, the issues of integration and analysis of data coming from diverse sources – with varying formats – are not covered under this model and requires a radical change in their approach.

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 RDF graph streams. In addition, it can also employ the static background information (static RDF datasets or ontologies) to reason upon the context of detected events. Thus, the events within a data stream are enriched with semantics, which in turn can lead to new applications that tackle the variety and heterogeneity of data sources.

The design of an efficient SCEP system requires carefully marrying the temporal operators with the RDF data model and an additional characteristic of event enrichment. Even though SCEP can be evolved from the common practice of stitching heterogeneous environments, a well organised query language is a vital part of SCEP: it not only allows users to specify known queries or patterns of events in an intuitive way, but also to showcase the expected answers of a query while hiding the implementation details.

While there does not exist a standard language for expressing continuous queries over RDF graph streams, a few options have been proposed. In particular, a first strand of research focuses on extending the scope of SPARQL to enable the stateless continuous evaluation of RDF triple streams. These query languages include CQELS [13], C-SPARQL [14], $\text{SPARQL}_{\text{Stream}}$ [15]: they are used to match query-defined graph patterns, aggregates and filtering operators against RDF triple streams, i.e. a sequence of RDF triples $\langle subject, predicate, object \rangle$, each associated with a timestamp τ. These languages do not provide any temporal pattern matching operator and thus cannot be classified under the SCEP languages.

The second strand of research focused on SCEP languages to extend SPARQL with stateful operators, where few options have been proposed [16,17,18]. EP-SPARQL [16] – with its expressive language and framework – is the primary player in this field with other works focusing on a subset of operators and functionalities. EP-SPARQL extends SPARQL with sequence constructs that allow temporal ordering over triple streams and its semantics are derived from the ETALIS language [19]. Although EP-SPARQL is a pioneering work in the field of SCEP, it suffers from following drawbacks:

– It works on a single stream model, and thus does not support the integration of multiple heterogeneous streams.
– It lacks explicit *Kleene+*, and event selection operators.
– Its definition of sequence operator and graph pattern matching operators are mixed. Thus, making it difficult to extend it for RDF graph streams, i.e., a sequence of RDF graph events, where each event (τ, G) contains an RDF graph G associated with a timestamp τ or an interval.

Considering these shortcomings, our contribution in this paper is twofold. First, we present a novel query language and system, called SPAseq, to enable complex event processing over heterogeneous sources using the RDF graph model. SPAseq covers the aforementioned shortcomings of existing languages and systems, and provides a unified language for SCEP over RDF graph streams, while introducing expressive explicit operators over heterogeneous streams. The use of explicit operators lets the users specify complex queries at high level and enables the appropriate implementation details and optimisations at the domain specific level. 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 CEP operators, such as *Kleene+*, disjunction, conjunction over events from heterogeneous streams.

As our second contribution, we provide an executional framework for SPAseq using a non-deterministic finite automata (NFA) model called NFA_{scep}. Automata-based techniques have become a *de facto* standard for temporal sequencing on traditional relational data models [6,7,20]. In particular, the NFA model offers higher expressiveness and less complexity as compared to its

deterministic counterparts. Hence, leveraging the well-established techniques, SPASEQ queries are compiled over equivalent NFAs, and a run-based technique [21] is used for the efficient evaluation of SPASEQ queries. Moreover, we employ various optimisation techniques on top of our system. It includes: indexing and partitioning of incoming streams by run id; pushing the stateful predicates and query windows; and lazy evaluation of the disjunction and conjunction operators.

Our main contributions are summarised as follows:

- We present the design and syntax of a novel SCEP query language, called SPASEQ, through intuitive examples.
- We provide the detailed semantics of SPASEQ and its main operators.
- We provide the NFA-based framework to efficiently compile and evaluate SPASEQ queries.
- We provide system and operator-level optimisation strategies for SPASEQ queries.
- Using real-world and synthetic datasets, we show the effectiveness of our optimisation strategies and our experimental evaluations show that they outperforms existing systems for the same use cases and datasets.

The rest of the paper is structured as follows. Section 2 presents the motivation of a new language and reviews the limitations of existing approaches. Section 3 presents the data model and syntax of SPASEQ. Section 4 presents intuitive examples of SPASEQ queries. Section 5 presents the semantics of the SPASEQ language. Section 6 provides a qualitative analysis of SPASEQ as compared with related query languages. Section 7 presents the details about the compilation of SPASEQ queries onto the NFA_{scep} model, the evaluation of NFA_{scep} automata and the design of the SPASEQ query engine. Section 8 presents the optimisations techniques used for the SPASEQ query engine. Section 9 provides experimental evaluations of the SPASEQ query engine.

2. Why A New Language?

In order 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.

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 fuel-based power source events, the second stream (S_2) provides the weather-related events, and the third stream (S_3) provides the storage-related source events that is attached to a renewable power source. 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 aforementioned three RDF graph streams S_1, S_2 and S_3. These are fed to an application that notifies the user to switch to renewable power source instead of fuel-based power source, if the system observes the following sequence of events: (**A**) the price of electricity generated by a fuel-based power source is greater than a certain threshold; (**B**) weather conditions are favourable for renewable energy production (one or more events); and (**C**) the price of storage source attached to the renewable power source is less than the fuel-based power source.*

UC 1 highlights the following main principles of a SCEP language:

- Since the RDF model is the corner-stone of SCEP, its features such as seamless integration of multiple heterogeneous streams (as described in **UC 1**) should be considered for the design of a SCEP language.
- The main aim of an SCEP language is to provide temporal operators on top of standard SPARQL operators. Thus, the list of temporal operators introduced for CEP over relational models [4,9, 22], such as sequencing, conjunction, disjunction, *Kleene+* and event selection strategies should be supported in a SCEP language.
- The SCEP language – by following the customary design of semantic stream processing languages such as CQELS, C-SPARQL – should provide operators to directly enrich events through a static background knowledge. For instance, a static background knowledge, such as user profiles and detailed information about the location of power sources, can further enrich the context in **UC 1**.

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

- The SCEP language should follow the principle of *genericity*, i.e. its design should be independent of the underlying execution model.
- 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 points underline the main requirements for a SCEP language. Herein, using them as a yardstick, we outline the limitations of existing languages.

2.2. Limitations of Existing Languages

Existing languages for RDF stream processing systems differ from each other in a wide range of aspects, which include the executional semantics, data models and targeted use cases. In this section, we adopt the same classification criteria as used in [23], and divide the systems into two classes: RDF Stream Processing (RSP) systems, and Semantic Complex Event Processing (SCEP) systems. Their details are discussed as follows.

2.2.1. RDF Stream Processing (RSP) Systems

The standardisation of the RSP is still an ongoing debate and the W3C RSP community group[1] is an important initiative in this context. Most of the RSP systems [13,14,15,24] inherit the processing model of Data Stream Management Systems (DSMSs), but consider a semantically annotated data model, namely RDF triple streams. Query languages for these systems are inspired from CQL [25], where a continuous query is composed from three classes of operators, namely stream-to-relation (S2R), relation-to-relation (R2R), and relation-to-stream (R2S) operators. C-SPARQL [14] and CQELS [13] are among the first contributions, and often cited as a reference in this field. They support timestamped RDF triples and queries are continuously updated with the arrival of new triples. The query languages for both systems extend SPARQL with operators such as `FROM STREAM` and `WINDOW` to select the operational streams, and the most recent triples within sliding windows. They also support the integration of background static data to further enrich the incoming RDF triples. Unlike the aforementioned systems, recently, we proposed a system called SPECTRA [26] to process RDF graph streams. It provides various system and operator level optimisations and continuously processes the standard SPARQL queries over RDF graph streams.

All the aforementioned systems and various others [15,24] are mainly developed as real-time monitoring systems: the states of the events are not stored to implement temporal pattern matching among a set of events. For the same reason, their query languages do not provide any operators for temporal pattern matching.

2.2.2. Semantic CEP Systems

Semantic CEP (SCEP) systems are evolved from the classical rule-based CEP systems, i.e. by integrating high-level knowledge representation and background static knowledge. To the best of our knowledge, EP-SPARQL is the only system that provides a unified language and executional framework for processing semantically enriched events with temporal ordering. The main building blocks of the EP-SPARQL language are represented by a set of four binary temporal operators: `SEQ`, `EQUALS`, `OPTIONAL-SEQ`, and `EQUALS-OPTIONAL`, which can be combined to express complex sequence patterns over RDF triple streams; each incoming triple-based event is associated with a time interval having start and end times.

Although EP-SPARQL is a pioneering work in the field of SCEP, it lacks various important features. These limitations are discussed as follows:

- *Multiple Heterogeneous Streams*: The EP-SPARQL data model is based on a single stream model. That is, a single RDF triple 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 an existing CEP system (ETALIS), where an RDF triple stream is mapped onto a Prolog object stream and language operators are mapped onto the underlying ETALIS language. Hence, its design is directly motivated from its underlying executional model, and extending it for the multiple streams model requires extensive overhauling of its semantics.

[1] https://www.w3.org/community/rsp/

Table 1
Properties of the Existing SCEP Systems

CEP Systems	Input Model	Operators	Available	Implementation
EP-SPARQL [16]	Triple Streams	Sequence, Conjunction, Disjunction, Optional	✓	
STARQL [17]	Triple Streams	Sequence	✗	
RSEP-QL [18]	Graph Streams	Sequence, Event Selection Strategies	✗	
CQELS-CEP [27]	Graph Streams	Sequence,Optional, Negation	✗	

– *Temporal Operators*: EP-SPARQL only supports a small subset of temporal operators. Operators such as *Kleene+* or event selection strategies are not supported. These operators are important for many applications where semantic noise is observed (more details are provided in Section 6.3). Moreover, the conjunction and disjunction operators in EP-SPARQL are inspired from SPARQL (UNION and AND). These operators do not provide the nesting over a set of events as described for CEP systems [5,22]. 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 operator 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 [28,29].

Apart from EP-SPARQL, recently, some other works also provide the intuition of SCEP. Some of these works are presented mainly for the purpose of theoretical analysis instead of practical implementation, while others take an approach for transforming queries over ontologies into relational ones, via ontology-based data access (OBDA) with temporal reasoning. Table 1 shows the properties of these systems. STARQL [30] uses the OBDA technique to determine Abox sequencing with a sorted first order logic on top of them. It provides simple formalism/mapping to SQL for sequence operators and all the other operators (such as *Kleene+*, conjunction, disjunction, event selection strategies) are not part of its framework. Furthermore, it is not a freely available system and it does not provide operators for explicitly referencing different points in time [17]. CQELS recently proposed in [27] the integration of sequence and path negation operators inspired from EP-SPARQL. However, its sequence clause is evaluated over a single stream and its syntax and semantics does not include event selection strategies. RSEP-QL [18] is a reference model to capture the behaviour of existing RSP solutions and to capture the semantics of EP-SPARQL's sequence operator. It is based on RDF graph model; however, its main focus is to capture the event selection strategies and other complex operators are currently not supported.

In this section, we pointed out various general requirements of an SCEP query language and the limitations of existing SCEP systems and query languages. Using the aforementioned discussion, we present the design of a new SCEP language in the subsequent sections.

3. The SPASEQ Query Language

Considering the shortcomings of EP-SPARQL and other languages, we propose a new language called SPASEQ. The design of SPASEQ is based on the following main principles: (1) support of an RDF graph stream model; (2) adequate expressive power, i.e. not only based on core SPARQL constructs but also including general purpose temporal operators (inspired from the common CEP operators); (3) genericity, i.e. independent of the underlying evaluation techniques; (4) clear separation between the temporal and RDF graph operators; (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, espe-

cially 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+*, disjunction, conjunction and event selection strategies over events from heterogeneous streams. In the following, we start with the data model over which SPAseq queries are processed and then provide the details regarding its syntax and semantics.

3.1. Data Model

In this section, we introduce the structural data model of SPAseq that captures the concept of RDF graph events: this serves as the basis of our query language. We use the RDF data model [31] to model an event. That is, we assume three pairwise disjoint and infinite sets of IRIs ($\mathcal{I}$), blank nodes ($\mathcal{B}$), and literals ($\mathcal{L}$). An RDF triple is a tuple $\langle s, p, o\rangle \in (\mathcal{I}\cup\mathcal{B})\times\mathcal{I}\times(\mathcal{I}\cup\mathcal{B}\cup\mathcal{L})$ and an RDF graph is a set of RDF triples. Based on this, the concepts of RDF graph event and stream are defined as follows.

Definition 1 *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 sets of RDF graph events defined as follows:

Definition 2 *An* **RDF graph event stream** $\mathcal{S}$ *is a possibly infinite set of RDF graph events such that, for any given timestamps* τ *and* τ'*, there is a finite amount of events occurring between them, and there is at most one single graph associated with any given timestamp.*

An RDF graph event stream can be seen as a sequence of chronologically ordered RDF graphs marked with timestamps. The constraints ensure that it is always possible to determine what unique event immediately precedes or succeeds a given timestamp. Without the first restriction, it would be possible to define a stream $\{(\frac{1}{n}, G) \mid n \neq 0 \text{ an integer}\}$ where there is no event immediately succeeding 0. In order to handle multiple streams, we identify each using an IRI and group them in a data model we call RDF *streamset*.

Definition 3 *A* **named stream** *is a pair* $(u, \mathcal{S})$ *where u is an IRI, called the* stream name*, and* $\mathcal{S}$ *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 paper, we simply use the terms *graph* for RDF graph, *event* for RDF graph event, *stream* for RDF graph stream, and *streamset* for RDF graph streamset.

Example 1 *Recall* ***UC*** *1, here we extend it with our data model. The first named stream* $(u_1, \mathcal{S}_1)$ *provides the events about the power-related sources from a house, the second named stream* $(u_2, \mathcal{S}_2)$ *provides the weather-related events for house, and the third named stream* $(u_3, \mathcal{S}_3)$ *provides the power storage-related events. Figure 1 illustrates the general structure of the events from each source. For instance, the named stream* $(u_1, \mathcal{S}_1)$ *can contain the following events:*

time	**graph**

10	`:H1 :loc :L1`
	`:H1 :pow :Pw1`
	`:Pw1 :source 'solar'`
	`:Pw1 :fare 5`
	`:Pw1 :watt 20`

15	`:H2 :loc :L2`
	`:H2 :pow :Pw2`
	`:Pw2 :source 'wind'`
	`:Pw2 :fare 6`
	`:Pw2 :watt 20`

...	...

3.2. 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 of SPARQL introduced in [32]. In particular, we rely on the notion of SPARQL graph pattern by considering operators AND, OPT, UNION, FILTER, and GRAPH.

Definition 4 *A SPAseq SELECT query is a tuple* $Q = (\mathcal{V}, \omega, \text{SeqExp})$*, where* $\mathcal{V}$ *is a set of variables,* ω *is a*

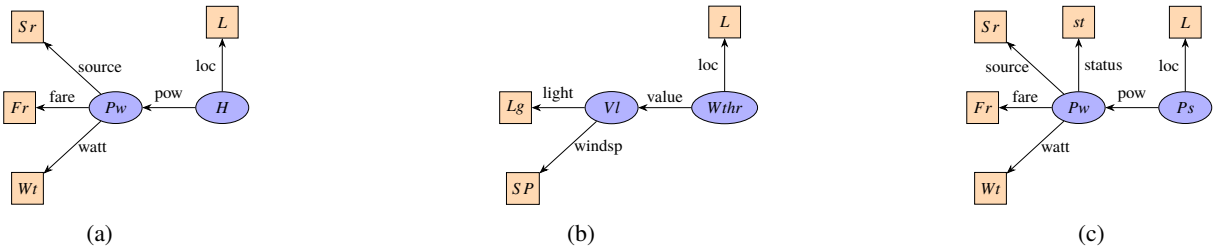

Figure 1. Structure of the Events from three Named Streams, (1a) $(u_1, \mathcal{S}_1)$ Power Event, (1b) $(u_2, \mathcal{S}_2)$ Weather Event, (1c) $(u_3, \mathcal{S}_3)$ Power Storage Event

duration, and SeqExp *is a* sequence expression *defined according to the following grammar:*

$$
\begin{aligned}
\text{SeqExp} &::= \mathit{Atom} \mid \text{SeqExp } \text{`;'} \mathit{Atom} \mid \text{SeqExp } \text{`,'} \mathit{Atom} \\
\mathit{Atom} &::= \mathit{GPM} \mid \mathit{GPM}\text{`+'} \mid \mathit{BOP} \\
\mathit{BOP} &::= \mathit{GPM}((\text{`\&'} \mid \text{`|'}))\ \mathit{BOP} \\
\mathit{GPM} &::= (u, P) \mid (u, P)\ \textbf{\textit{Graph}}\ (u, P_D)
\end{aligned}
$$

where u is an IRI, P is a SPARQL graph pattern, (u, P) is called a graph pattern matching expression *(GPM), and P_D is a SPARQL graph pattern defined for a static RDF graph, i.e. an external knowledge-base.*

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

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

The concrete syntax of SPAseq is illustrated in Query 1 which includes syntactic sugar that is close to SPARQL. It contains three GPM expressions each identified with a name (`A`, `B`, and `C`), which allows one to concisely refer to GPMs and to the named streams. These names are employed by the sequence expression to apply various temporal operators. The sequence expression in Query 1 is presented at *line 9*; the streams are described at *lines 3-5*; the GPM expressions on these streams start at *lines 11, 19 and 27*.

One of the main property of the SPAseq language is depicted in Query 1, i.e. 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. We start with the brief description of unary operator ({'+'}), the event selection strategies ({';', ','}) and binary operators ({'&', '|'}). The details of these operators are covered during the description of their semantics in Section 5. Furthermore, we also present the *Graph* operator for the SPAseq query language.

3.2.1. SPAseq Unary Operators

The sequence expression SeqExp in SPAseq is used to determine the sequence between the events matched to the graph pattern P. The symbol {'+', } corresponds to the *Kleene+* operator. It determines the occurrence of one or more events of the same kind. This means a series of events can be matched using this operator.

3.2.2. SPAseq Event Selection Strategies

In general, the *sequence temporal patterns* between events detect the occurrence of an event *followed-by* another. However, a deeper look reveals that it can represent various different circumstances using different event selection strategies [4]. These selection strategies overload the sequence operator with the constraints to define how to select the relevant events from an input stream, while mixing relevant and irrelevant events. The symbols {';', ','} are binary operators which describe the interpretations of the sequence between events. An event G_e^i matched to the graph pattern P_i followed-by an event G_e^j matched to the graph

pattern 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 any named stream 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.

3.2.3. SPAseq Binary Operators

Conjunction and disjunction defined over the event streams constitute the binary operators. In SPAseq, these operators are introduced within the sequence expression through symbols (‘**&**’) and (‘**|**’) respectively. They provide the intuitive way of determining if a set of events happen at the same time (conjunction) or at least one event among the set of events happens (disjunction).

Example 2 *Consider the SPAseq Query 1, which illustrates the* ***UC 1****. The sequence expression* `SEQ(A; B+, C)` *illustrates that the query returns a match: if an event of type* `A` *defined on a stream* `S1` *matches the GPM expression* `A` followed-by *one or more events (using operators (‘**;**’) and (‘**+**’)) from stream* `S2` *that match the GPM expression* `B`*, and finally* immediately followed-by *(using operator (‘**,**’)) an event from stream* `S3` *that matches the GPM expression* `C`*. Notice that a GPM expression mainly utilises SPARQL graph patterns for the evaluation of each event.*

3.3. SPAseq Graph, Window and Stream Operator

The combination of streaming information in the form of RDF graph streams and other information from the static knowledge base can lead to novel semantics and information rich CEP. The *Graph* operator in SPAseq is designed to take advantage of static information available in the form of an RDF graph. Thus, a graph pattern P_D defined over the static RDF graph G_D is first evaluated and then the results are matched with the incoming stream $\mathcal{S}$. This leads to a SCEP system, where detailed information regarding a context can be revealed with the help of already available static datasets.

In SPAseq, the sequence expression is defined over a streamset. Thus, we use the `FROM STREAM` clause to define a set of streams. For instance, in Query 1 we use three streams identified as `S1`, `S2` and `S3` (*lines 3-5*). These stream names are used within the defined GPM expressions. Furthermore, since the sequence over a set of events is constrainted by the temporal window, we use the `WITHIN` clause to define the temporal windows (*line 2* in Query 1). In SPAseq, windows can be defined in seconds, minutes and hours. For instance, in Query 1 we use a `60 MINUTES` window.

4. SPAseq By Examples

In this section, we provide a list of use cases supported by SPAseq, while highlighting its SPARQL-based and temporal operators. Each example showcases a specific operator supported by SPAseq.

```
1  PREFIX  pred: <http://example/>
2  SELECT  ?company ?p1 ?p2 ?p3 ?p4 ?p5 ?p6 ?p7 ?vol1 ?vol2
       ?vol3 ?vol4 ?vol5 ?vol6 ?vol7
3  WITHIN 60 MINUTES
4  FROM STREAM S1 <http://stockmarket.org/stocks/google>
5
6  WHERE {
7
8    SEQ (A, B, C, D, E, F, G)
9
10   DEFINE GPM A ON S1 {
11     ?company pred:price ?p1.
12     ?company pred:volume ?vol1.
13   }
14
15   DEFINE GPM B ON S1 {
16     ?company pred:price ?p2.
17     ?company pred:volume ?vol2.
18     FILTER (?p2 > ?p1)
19   }
20
21   DEFINE GPM C ON S1 {
22     ?company pred:price ?p3.
23     ?company pred:volume ?vol3.
24     FILTER (?p3 < ?p2 && ?p3 > ?p1).
25   }
26
27   DEFINE GPM D ON S1 {
28     ?company pred:price ?p4.
29     ?company pred:volume ?vol4.
30     FILTER (?p4 > ?p2).
31   }
32
33   DEFINE GPM E ON S1 {
34     ?company pred:price ?p5.
35     ?company pred:volume ?vol5.
36     FILTER (?p5 < ?p4 && ?p5 > ?p3).
37   }
38
39   DEFINE GPM F ON S1 {
40     ?company pred:price ?p6.
41     ?company pred:volume ?vol6.
42     FILTER (?p6 > ?p5 && ?p6 < ?p4).
43   }
44
45   DEFINE GPM G ON S1 {
46     ?company pred:price ?p7.
47     ?company pred:volume ?vol7.
48     FILTER (?p7 < ?p6 && ?p7 > ?p5).
49   }
50 }
```

Query 2: Head and Shoulders Pattern: SPAseq query

UC 2 (Head and Shoulders Classification) *The Head and shoulders pattern consists of the following peaks:*

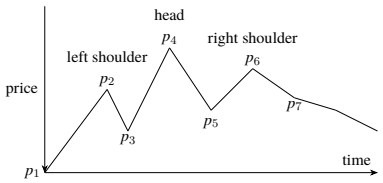

Figure 2. Head and shoulders pattern for the **UC** 2

(i) the left shoulder that is formed by an uptrend; (ii) the head being a peak higher than the left shoulder; (iii) the right shoulder that shows an increase but fails to take out the peak value of the head. Figure 2 shows such a pattern for the stock prices of a company.

Head and shoulders classification is a well-known pattern to predict the future development of varying values and belongs to the family of reversal patterns [33]. It is an extension of the V-shaped pattern [9], i.e. the sequence of values that go down to a lower value, then rising up to a higher value, which was higher than the starting value.

```
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 "harbour".
    ?vessel pred:direction ?dir1.
    FILTER (?s1 > 0)
  }

  DEFINE GPM B ON S1 {
    ?vessel pred:speed ?s2.
    ?vessel pred:location ?l.
    ?vessel pred:direction ?dir1.
    FILTER (?s2 > ?s1)
  }

  DEFINE GPM C ON S1 {
    ?vessel pred:speed 0.
    ?vessel pred:location "fishingarea".
    ?vessel pred:direction ?dir3.

    GRAPH <http://harbour.org/db> {
      ?vessel :name ?n.
      ?vessel :operatedBy ?company.
      ?company :name ?cname.
    }
  }
}
```

Query 3: Trajectory Classification: SPAseq query

The application of the head and shoulders pattern ranges from stock analysis, weather prediction to trajectory classification.

Query 2 shows a SPAseq query for such pattern over the stream of Google stocks, where a set of GPM expressions are used to determine the sequence over the stock price values. The `SELECT` expression provides the projection of various variables within the GPM expressions, while the GPM expressions utilise the `?company` variable to select the company mappings, and its corresponding volume and price mappings. The head and shoulders pattern in Query 2 can also be spiced up with the disjunction operator to evaluate the occurrence of negative head and shoulders.

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

A SPAseq query to determine the trajectory of fishing boats is described in Query 3. It represents the following sequence: **A**: vessel leaves the harbour, **B**: vessel travels by keeping 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 Query 3 is the *Graph* operator to join the event data with static knowledge base. That is, using an external knowledge base, the query extracts the name (`?n`) and company name (`?cname`) of the vessels that follow the sequence defined in the sequence expression.

UC 4 (Inventory Management) *Consider an inventory management system monitoring the status (surgical usage, recycling, etc.) of equipments in a hospital by using various RFID sensors. We can define a complex event by monitoring if a surgical tool is washed/recycled and is put back into the use following the process of either disinfection or sterilisation.*

In the aforementioned use case, RFID generated events are used to track the status of a product/equipment. Furthermore, we need to construct a new event composed of the detected events.

Query 4 presents the **UC 4**, and it consists of four GPM expressions. The first GPM expression (`GPM A ON S1`) determines the recycling status of an instrument, the second and third GPM expressions (`GPM B ON S1` and `GPM C ON S1`) utilise the same variable for the instrument (`?inst`) to determine if it has been either disinfected or sterilized, and the fourth GPM expression determines the status of the instrument, i.e. if it has been used or not. The sequence expression

(SEQ(A; B|C; D)) orchestrates the matching of the GPM expressions. The disjunction operator ('|') in the sequence expression makes sure that the sequence is only matched if there are events of type B or C –i.e. the status of an instrument is "disinfected" or "sterilized" – between A and D.

```
PREFIX  pred: <http://example/>

CONSTRUCT S2 <http://hospital.org/newStream> {
  ?inst pred:InRoom ?r3.
  ?inst pred:name ?n1.
}
WITHIN 60 MINUTES
FROM STREAM S1 <http://hospital.org/instruments>

WHERE {

  SEQ (A; B|C; D)

  DEFINE GPM A ON S1 {
    ?inst pred:name ?n1.
    ?inst pred:status "recycled"@en.
  }

  DEFINE GPM B ON S1 {
    ?inst pred:status "disinfected"@en.
  }

  DEFINE GPM C ON S1 {
    ?inst pred:status "sterilized"@en.
  }

  DEFINE GPM D ON S1 {
    ?inst pred:InRoom ?r3.
    ?inst pred:name ?n1.
    ?inst pred:status "can use"@en.
  }
}
```

Query 4: Inventory Management: SPAseq query

The CONSTRUCT clause in Query 4 shows the composition of new events from the detected ones. That is, new RDF graph events can be constructed/generated if the defined sequence is matched with the incoming events. SPAseq employs the standard CONSTRUCT expression from SPARQL to create new graph events from the matched mappings. The set of constructed events takes the form of a stream (S2 in Query 4), and they can either be transmitted to the defined sink (an application) or can be reused within the defined query. Note that the syntax and implementation of SPAseq supports the CONSTRUCT clause. However, for the sake of brevity, during the discussion of SPAseq semantics, we focus on SELECT SPAseq queries.

5. Formal Semantics of SPAseq

To formally define the semantics of SPAseq queries, we reuse concepts from the semantics of SPARQL as defined in [32]. A *mapping* is a partial function from a set of variables to RDF terms ($\mathcal{B} \cup \mathcal{I} \cup \mathcal{L}$). The *domain* of a mapping μ, denoted $\text{dom}(\mu)$, is the set of variables that have an image via μ. We say that two mappings μ and μ' are *compatible* if they agree on all shared variables, i.e. if $\mu(x) = \mu'(x)$ for all $x \in \text{dom}(\mu) \cap dom(\mu')$. For a graph pattern P, we denote by $\text{vars}(P)$ the set of variables appearing in P and $\mu(P)$ is the graph pattern obtained by replacing each variable $v \in \text{vars}(P)$ by $\mu(v)$ whenever defined.

We repeat the definitions of join ($\bowtie$), union ($\cup$), minus ($\setminus$), left outer-join ($\leftouterjoin$) and evaluation of graph patterns as in [32].

Definition 5 *Let Ω_1 and Ω_2 be sets of mappings:*

$\Omega_1 \bowtie \Omega_2 = \{\mu_1 \cup \mu_2 \mid \mu_1 \in \Omega_1, \mu_2 \in \Omega_2, \mu_1 \text{ and } \mu_2 \text{ compatible}\}$
$\Omega_1 \cup \Omega_2 = \{\mu \mid \mu \in \Omega_1 \text{ or } \mu \in \Omega_2\}$
$\Omega_1 \setminus \Omega_2 = \{\mu_1 \in \Omega_1 \mid \text{for all } \mu_2 \in \Omega_2,\ \mu_1 \text{ and } \mu_2 \text{ not compatible}\}$
$\Omega_1 \leftouterjoin \Omega_2 = (\Omega_1 \bowtie \Omega_2) \cup (\Omega_1 \setminus \Omega_2)$

Definition 6 *Let t be a triple pattern, P, P_1, P_2 graph patterns and G an RDF graph, then the evaluation $[\![\cdot]\!]_G$ is recursively defined as follows:*

$[\![t]\!]_G = \{\mu \mid dom(\mu) = \text{vars}(t) \text{ and } \mu(t) \in G\}$
$[\![P_1 \text{ AND } P_2]\!]_G = [\![P_1]\!]_G \bowtie [\![P_2]\!]_G$
$[\![P_1 \text{ UNION } P_2]\!]_G = [\![P_1]\!]_G \cup [\![P_2]\!]_G$
$[\![P_1 \text{ OPTIONAL } P_2]\!]_G = [\![P_1]\!]_G \leftouterjoin [\![P_2]\!]_G$
$[\![P \text{ FILTER } R]\!]_G = \{\mu \in [\![P]\!]_G \mid \mu(R) \text{ is true}\}$

In this section, we define the semantics of SPAseq in a bottom-up manner, where we start with the semantics of GPM expressions by integrating the temporal aspects of events and streams. Note that, for the sake of brevity, we show the evaluation of GPM expressions over a streamset and the aspects of evaluating *Graph* operator (over RDF dataset) within GPM expressions are discussed later. This will aid us in highlighting the decisions we took to define the semantics of SPAseq operators.

5.1. Evaluation of Graph Pattern Matching Expressions

In SPAseq, Graph Pattern Matching expressions are evaluated against a streamset over a finite time interval that "tumbles" as time passes. Consequently, we constrain the evaluation function to a temporal boundary (i.e. a window), with a start time (τ_b) and an end time (τ_e). In addition, we use the notation $\Sigma(u)$ to select a stream of name u from a streamset, such that

$$\Sigma(u) = \begin{cases} \mathcal{S} & if \ \ (u, \mathcal{S}) \in \Sigma \\ \varnothing & otherwise \end{cases}$$

The evaluation of a GPM expression is defined as follows.

Definition 7 *The **evaluation of a GPM expression** (u,P) over the streamset Σ for the time boundaries $[\tau_b,\tau_e]$ is:*

$$[\![(u,P)]\!]_{\Sigma}^{[\tau_b,\tau_e]} = \{(\tau,[\![P]\!]_G) \mid (\tau,G) \in \Sigma(u) \wedge \tau_b \leq \tau \leq \tau_e) \wedge [\![P]\!]_G \neq \emptyset\}$$

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 matched event. In the absence of a match, no results are returned for the considered timestamp.

Example 3 *Consider a GPM expression $(u_1,P_1) := (u_1,\{(?h,pow,?p),(?h,loc,?l)\})$ and a power-related named stream $(u_1,\mathcal{S}_1) \in \Sigma$ with events as follows:*

time	graph
10	:H1 :pow :Pw1 :H1 :loc :L1
15	:H2 :pow :Pw2 :H2 :loc :L2
25	:H3 :pow :Pw3 :H3 :loc :L3
...	...

The evaluation of (u_1,P_1) over Σ for the time boundaries [5,15], i.e. $[\![(u_1,P_1)]\!]_{\Sigma}^{[5,15]}$, is described as follows:

time	?h	?p	?l
10	:H1	:Pw1	:L1
15	:H2	:Pw2	:L2

Notice that since the end time of the window is restricted at $\tau = 15$, only the events at $\tau = 10$ and $\tau = 15$ are included in the result. The event at $\tau = 25$ is outside the window and thus is not included in the results.

In order to define the semantics of sequence expressions, we introduce the notion of BOp expressions, for conjunction and disjunction operators, defined as follows:

Definition 8 *A* BOp expression *is either a GPM expression or an expression containing exclusively binary operators '**&**' and '|'.*

Consequently, a BOp expression does not contain *Kleene+*, followed-by, or immediately followed-by operators.

5.2. Evaluation of Binary Operators

Herein, we define the semantics of binary operators provided for SPAseq, i.e. conjunction and disjunction of events.

Definition 9 *Given two BOp expressions Ψ_1,Ψ_2 the **evaluation of the conjunction** operator over the streamset Σ and for the time boundaries $[\tau_b,\tau_e]$ is defined as follows:*

$$[\![\Psi_1 \,\&\, \Psi_2]\!]_{\Sigma}^{[\tau_b,\tau_e]} = \left\{ \begin{array}{l} (\tau, X \bowtie Y) \mid (\tau,X) \in [\![\Psi_1]\!]_{\Sigma}^{[\tau_b,\tau_e]} \wedge \\ (\tau,Y) \in [\![\Psi_2]\!]_{\Sigma}^{[\tau_b,\tau_e]} \wedge X \bowtie Y \neq \emptyset \end{array} \right\}$$

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 4 *Consider the following, a GPM expression $(u_1,P_1) := (u_1,\{(?h,pow,?p),(?h,loc,?l)\})$ and a power-related named stream $(u_1,\mathcal{S}_1) \in \Sigma$ as follows:*

time	graph
10	:H1 :pow :Pw1 :H1 :loc :L1
25	:H2 :pow :Pw2 :H2 :loc :L2
...	...

Now consider a GPM expression $(u_2,P_2) := (u_2,\{(?w,value,?v),(?w,loc,?l)\})$ and a weather-related named stream $(u_2,\mathcal{S}_2) \in \Sigma$ as follows:

time	graph
10	:W1 :value :Vl1 :W1 :loc :L1
20	:W2 :value :Vl2 :W2 :loc :L1
...	...

*The evaluation of the conjunction operator over the aforementioned GPM expressions and named streams ($[\![(u_1,P_1)$ **&** $(u_2,P_2)]\!]_{\Sigma}^{[10,25]}$) for the time boundaries [10,25] will result in the following sets of mappings.*

time	?h	?p	?l	?w	?v
10	:H1	:Pw1	:L1	:W1	:Vl1

We now define the disjunction operator. It detects the occurrence of events that match to a GPM expression within the set of defined ones.

Definition 10 *Given two BOp expressions Ψ_1 and Ψ_2, the **evaluation of the disjunction** operator over the streamset Σ and for the time boundaries $[\tau_b, \tau_e]$ is defined as follows:*

$$[\![\Psi_1 \mid \Psi_2]\!]_{\Sigma}^{[\tau_b,\tau_e]} = [\![\Psi_1]\!]_{\Sigma}^{[\tau_b,\tau_e]} \cup [\![\Psi_2]\!]_{\Sigma}^{[\tau_b,\tau_e]}$$

Example 5 *Consider the two GPM expressions (u_1, P_1) and (u_2, P_2), and the two named streams $(u_1, \mathcal{S}_1), (u_2, \mathcal{S}_2) \in \Sigma$ from Example 4.*

The evaluation of the disjunction sequence operator for the sequence expression $[\![((u_1, P_1) \mid (u_2, P_2)))]\!]_{\Sigma}^{[10,25]}$, and for the time boundaries [10,25] is as follows:

time	?h	?p	?l	?w	?v
10	:H1	:Pw1	:L1		
10			:L1	:W1	:Vl1
15			:L1	:W1	:Vl1
20			:L1	:W2	:Vl2
25	:H2	:Pw2	:L2		

Notice that the disjunction operator may generate several sets of mappings for the same timestamp, as we can see at time 10 in the example.

5.3. Evaluation of Event Selection Operators

The event selection operators, i.e. *followed-by* and *immediately followed-by*, determine how a matched event follows the other. Interpreting sequence operators has been shown to be subject to design decisions, as shown by [4,18]. Here we present two possible interpretations of the *followed-by* operator. These interpretations can be classified as *skip-till-next* and *skip-till-any* according to [4]. While the latter is relatively easy to define and understand, the former leads to more efficient implementations.

Let σ be a sequence with a set of GPM expressions and binary/unary operators. The evaluation of the *followed-by* operator according to the skip-till-any interpretation is defined as follows:

Definition 11 *Given a sequence σ and an BOp expression Ψ, the **evaluation of the followed-by (;)** sequence operator in the skip-till-any configuration over a streamset Σ for the time boundaries $[\tau_b, \tau_e]$ is defined as follows:*

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

From the above definition, the evaluation of the *followed-by* operator is simply the join between the mapping sets from σ and the GPM expression. Its evalution is explained in the following example.

Example 6 *Consider the following, a GPM expression $(u_1, P_1) := (u_1, \{(?h, pow, ?p), (?h, loc, ?l)\})$ and a power-related named stream $(u_1, \mathcal{S}_1) \in \Sigma$ as follows:*

time	graph
10	:H1 :pow :Pw1 :H1 :loc :L1
15	:H2 :pow :Pw2 :H2 :loc :L1
...	...

A GPM expression as follows:

$$(u_2, P_2) := (u_2, \{(?w, value, ?v), (?w, loc, ?l)\})$$

and a weather-related named stream $(u_2, \mathcal{S}_2) \in \Sigma$ as follows:

time	graph
15	:W1 :value :Vl1 :W1 :loc :L2
20	:W1 :value :Vl1 :W1 :loc :L1
25	:W2 :value :Vl2 :W2 :loc :L1
...	...

Then for the evaluation of the followed-by *operator in the skip-till-any configuration on these GPM expressions for the time boundaries [10,25], i.e. $[\![(u_1, P_1);(u_2, P_2)]\!]_{\Sigma}^{[10,25]}$, we have the mappings as given below.*

time	?h	?p	?l	?w	?v
20	:H1	:Pw1	L1	:W1	:Vl1
25	:H1	:Pw1	L1	:W2	:Vl2
20	:H2	:Pw2	L1	:W1	:Vl1
25	:H2	:Pw2	L1	:W1	:Vl1

Notice that there are four matches sequences: both the first and seconds events (at $\tau = 10$ and $\tau = 15$) from the power-related named stream matched with both events ($\tau = 20$ and $\tau = 25$) in the weather-related named stream. This is due to the skip-till-any nature of the followed-by operator and all the possible combinations of matches are produced.

The following definition shows the evaluation of the followed-by operator under the skip-till-next configuration.

Definition 12 *Given a sequence σ and an BOp expression Ψ, the **evaluation of the followed-by (;)** sequence operator in the skip-till-next configuration over a streamset Σ for the time boundaries $[\tau_b, \tau_e]$ is defined as follows:*

$$[\![\sigma;\Psi]\!]_{\Sigma}^{[\tau_b,\tau_e]} = \left\{ \begin{array}{c} (\tau, X \bowtie Y) \mid \exists \tau', (\tau', X) \in [\![\sigma]\!]_{\Sigma}^{[\tau_b,\tau_e]} \wedge \tau' < \tau \wedge \\ (\tau, Y) \in [\![\Psi]\!]_{\Sigma}^{[\tau_b,\tau_e]} \wedge X \bowtie Y \neq \emptyset \\ \forall \tau'' \forall Z \left((\tau'', Z) \in [\![\Psi]\!]_{\Sigma}^{[\tau_b,\tau_e]} \wedge (\tau' < \tau'' < \tau) \right) \Rightarrow X \bowtie Z = \emptyset \end{array} \right\}$$

Example 7 *Consider the GPM expressions (u_1, P_1), (u_2, P_2) and the streams from Example 6. Then for the evaluation of the* followed-by *operator in the skip-till-next configuration on these GPM expressions for the time boundaries [10,25], i.e. $[\![(u_1, P_1);(u_2, P_2)]\!]_{\Sigma}^{[10,25]}$, we have the mappings as given below.*

time	*?h*	*?p*	*?l*	*?w*	*?v*
20	*:H1*	*:Pw1*	*L1*	*:W1*	*:Vl1*
20	*:H2*	*:Pw2*	*:L1*	*:W1*	*:Vl1*

Due to the skip-till-next configuration of the followed-by *operator, there are only two matches in the aforementioned example. Both events (at $\tau = 10$ and $\tau = 15$) from the power-related stream match with just one event (at $\tau = 20$) from the weather-related stream. Nonetheless, the semantics provides a set of results for each timestamp from the second stream even if that set is empty. As already pointed out after Example 4, this is a desirable feature for the definition of the immediately-followed-by operator.*

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

Definition 13 *Given a sequence σ and a BOp expression Ψ, 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:*

$$[\![\sigma, \Psi]\!]_{\Sigma}^{[\tau_b,\tau_e]} = \left\{ \begin{array}{c} (\tau, X \bowtie Y) \mid \exists \tau', (\tau', X) \in [\![\sigma]\!]_{\Sigma}^{[\tau_b,\tau_e]} \wedge \\ (\tau, Y) \in [\![\Psi]\!]_{\Sigma}^{[\tau_b,\tau_e]} \wedge \tau' < \tau \wedge X \bowtie Y \neq \emptyset \wedge \\ \forall u \forall \tau'' \forall G \left((\tau'', G) \in \Sigma(u) \wedge \tau'' > \tau' \right) \Rightarrow \tau'' \geq \tau \end{array} \right\}$$

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 8 *Consider the GPM expressions and the named streams defined in Example 6. Then the evaluation of the* immediately followed-by *operator $([\![(u_1, P_1),(u_2, P_2)]\!]_{\Sigma}^{[10,25]})$, for time boundaries [10,25], will results in a single sequence match with the following mappings.*

time	*?h*	*?p*	*?l*	*?w*	*?v*
20	*:H2*	*:Pw2*	*:L1*	*:W1*	*:Vl1*

This is due to the strict ordering of the immediately followed-by operator. *That is, for first event (at $\tau = 10$) in power-related stream, there is no immediately followed-by event in the weather-related stream.*

5.4. Evaluation of Kleene+ Operator

We now move towards the definitions of the unary operator, namely *Kleene+*. We first define its semantics in a standalone manner and then recursively define it with the help of sequence σ.

Definition 14 *The **evaluation of the standalone Kleene+ operator** over the streamset Σ and for the time boundaries $[\tau_b, \tau_e]$ is defined using auxiliary constructs $\cdot^k$ for integers $k > 0$ 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. Notice that the evaluation will not only match the longest sequence of match-

ing patterns, but will also provide results for the shorter sequences (using *followed-by* operator (**;**) with the skip-till-next configuration). The case of the *Kleene+* operator with sequence σ using additional *followed-by* and *immediately followed-by* is defined as follows:

Definition 15 *Let $\bullet$ denote either , or ;. Given a sequence expression σ, the **evaluation of the Kleene+ operator in a sequence** over the streamset Σ and for the time boundaries $[\tau_b, \tau_e]$ are defined as follows:*

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

Example 9 *Consider the following, a GPM expression $(u_1, P_1) := (u_1, \{(?h, pow, ?p), (?h, loc, ?l)\})$ and a power-related named stream $(u_1, \mathcal{S}_1) \in \Sigma$ as follows:*

time	**graph**
10	`:H1 :pow :Pw1` `:H1 :loc :L1`
25	`:H2 :pow :Pw2` `:H2 :loc :L2`
...	...

A GPM expression

$$(u_2, P_2) := (u_2, \{(?w, value, ?v), (?w, loc, ?l)\})$$

and a weather-related named stream $(u_2, \mathcal{S}_2) \in \Sigma$ as follows:

time	**graph**
15	`:W1 :value :Vl1` `:W1 :loc :L1`
20	`:W2 :value :Vl2` `:W2 :loc :L1`
...	...

The evaluation of the sequence $[\![(u_1, P_1);(u_2, P_2)+]\!]_{\Sigma}^{[10,20]}$ with the Kleene+ and followed-by (skip-till-next) operators for the time boundaries [10,20] is as follows:

time	*?h*	*?p*	*?l*	*?w*	*?v*
15	*:H1*	*:Pw1*	*:L1*	*:W1*	*:Vl1*
20	*:H1*	*:Pw1*	*:L1*	*:W2*	*:Vl2*

Notice that the Kleene+ operator collects one or more matches for (u_2, P_2) from the weather-related named stream.

5.5. The Graph Operator

The *Graph* operator within GPM expressions allows one to query both static data and streams. In the previous sections, we omitted this construct because it needlessly makes the notations cumbersome: it would require adding an RDF dataset (in addition to a streamset) as a parameter of the evaluation function.

However, for completeness, we present the definition of the evaluation of the *Graph* operator. Now similarly to the Definition 7, we use a function $\Gamma(u)$ to select an RDF graph of name u from an RDF dataset D, such that:

$$\Gamma(u) = \begin{cases} \varnothing & \text{if } u \text{ is not a graph name in } D \\ G_D & \text{if } (u, G_D) \in D \end{cases}$$

Definition 16 *Let D be an external RDF dataset and (v, P_D) be a graph pattern. Let (u, P) be the GPM expression defined over the streamset Σ, and $[\tau_b, \tau_e]$ be the time boundaries. The evaluation of the GPM expression and the Graph operator is defined as follows:*

$$[\![(u,P)\ \texttt{Graph}\ (v,P_D)]\!]_{(\Sigma,\ D)}^{[\tau_b,\tau_e]} = \{(\tau, [\![P]\!]_G \bowtie [\![P_D]\!]_{G_D}) \mid \exists \tau(\tau, G) \in \Sigma(u) \quad \wedge \exists G_D \in \Gamma(v) \wedge \tau_b \leq \tau \leq \tau_e\}$$

The *Graph* operator provides a useful functionality in the context of SCEP: it allows static knowledge to be considered while using the graph structure of the incoming events.

Example 10 *Consider the same GPM expression (u_1, P_1) and power-related named stream $(u_1, \mathcal{S}_1) \in \Sigma$ presented in Example 9. Now consider a graph pattern $(u_D, P_D) = (u_D, \{(?h, owner, ?n), (?h, address, ?a)\})$ defined over an external RDF graph D as follows:*

G_D

`:H1 :owner :john`
`:H1 :address :paris`
`:H2 :owner :smith`
`:H2 :address :lyon`

The evaluation of the GPM expression and the Graph operator $[\![(u_1, P_1)\ Graph\ (u_D, P_D)]\!]_{(\Sigma,\ D)}^{[10,20]}$ is as follows:

time	*?h*	*?p*	*?l*	*?n*	*?a*
10	*:H1*	*:Pw1*	*:L1*	*:john*	*:paris*

5.6. Evaluation of SPAseq Queries

In the previous sections, we outline the semantics of temporal operators and *Graph* operator of the SPAseq query language. Herein, to sum it up, we present the evaluation of complete SPAseq SELECT queries.

Definition 17 *Let Ω be a mapping set, $\pi_{\mathcal{V}}$ be the standard SPARQL projection on the set of variables $\mathcal{V}$, and ω be the duration of the window. The evaluation of SPAseq* SELECT *query $Q = (\mathcal{V}, \omega, \text{SeqExp})$ issued at time t, over the streamset Σ is defined as follows:*

$$[\![Q]\!]^t_\Sigma = \bigcup_{k\in\mathbb{N}} \left\{ \begin{array}{c} (\tau, \pi_{\mathcal{V}}(\Omega)) \mid \\ (\tau,\Omega) \in [\![\text{SeqExp}]\!]^{[t+k\cdot\omega, t+(k+1)\cdot\omega]}_\Sigma \end{array} \right\}$$

where

$$\pi_{\mathcal{V}}(\Omega) = \left\{ \begin{array}{c} \mu_1 \mid \exists \mu_2 : \mu_1 \cup \mu_2 \in \Omega \wedge dom(\mu_1) \subseteq \mathcal{V} \wedge \\ dom(\mu_2) \cap \mathcal{V} = \varnothing \end{array} \right\}$$

The evaluation of the SPAseq queries follows 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 11 *Recall the two GPM expressions from Example 6, $(u_1, P_1) := (u_1, \{(?h, pow, ?p), (?h, loc, ?l)\})$ and $(u_2, P_2) := (u_2, \{(?w, value, ?v), (?w, loc, ?l)\})$. Now consider the power-related and weather-related named streams $(u_1, \mathcal{S}_1)$, $(u_2, \mathcal{S}_2) \in \Sigma$ respectively as follows:*

time	graph
10	`:H1 :pow :Pw1` `:H1 :loc :L1`
25	`:H2 :pow :Pw2` `:H2 :loc :L2`
...	...

time	graph
15	`:W1 :value :Vl1` `:W1 :loc :L1`
40	`:W2 :value :Vl2` `:W2 :loc :L2`
...	...

Then the evaluation of a SPAseq query

$$Q = (\{?h, ?p, ?v\}, 50, ((u_1, P_1)\ ;\ (u_2, P_2)))$$

with the followed-by operator using skip-till-next configuration at time $\tau = 20$ over the streamset Σ, i.e. $[\![Q]\!]^{20}_\Sigma$, can be described as follows:

time	*?h*	*?p*	*?v*
15	*:H1*	*:Pw1*	*:Vl1*

In this section, we presented the detailed semantics of SPAseq operators. The processing section presents how SPAseq can be extended for the new operators. The implementation details of SPAseq operators are provided in Section 7.

5.7. A Note on the Extension of SPAseq

As discussed earlier, the design of SPAseq encouraged the extensibility of the language with new operators. Herein, we present two operators and discuss how they can be integrated into the SPAseq query model and their effects on the semantics of SPAseq.

5.7.1. The case of Negation Operator

Negation is a unary operator and is used to describe the non-occurrence of certain events. For example, we can extend **UC 4** to determine the following sequence: **A**: *A surgical tool is washed/recycled* **B**: *The surgical tool is not disinfected* **C**: *The surgical tool is put back into use*. Let '**!**' denotes a negation operator, then the Query 5 presents a SPAseq query with negation operator over GPM expression **B**.

The standalone semantics of the negation operator can easily be defined. However, discrepancies arise when it is used with the *followed-by* and *immediately followed-by* operators. For instance, the evaluation of the negation operator for a GPM expression (u, P) over a streamset Σ and the time boundaries $[\tau_b, \tau_e]$ can be described as follows:

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

Thus, for each event that does not match with the GPM expression, the evaluation of the negation operator returns an empty set associated with a timestamp. The use of the negation operator in conjunction with the *followed-by* operator results in discrepancies, where it is difficult to differentiate between the results of a negation operator and the results of a simple GPM evaluation in case of non-occurrence of an event. Therefore, the negation operator requires a new structure such that we can differentiate between the empty set from the evaluation of GPM expressions and the GPM expressions with the 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

$$? \bowtie \Omega = \Omega \bowtie ? = \Omega$$

An identity element from a commutative monoid family can be employed to showcase the aforementioned behaviour of the negation operator.

```
PREFIX  pred: <http://example/>
CONSTRUCT S2 <http://hospital.org/newStream> {
 ?inst pred:InRoom ?r3.
 ?inst pred:status "non-disinfection"@en.
 ?inst pred:name ?n1.
 }
 WITHIN 60 MINUTES
 FROM STREAM S1 <http://hospital.org/instruments>

  WHERE {

     SEQ (A, B!, C)

     DEFINE GPM A ON S1 {
     ?inst pred:name ?n1.
     ?inst pred:status "recycled"@en.
    }

     DEFINE GPM B ON S1 {
      ?inst pred:status "disinfected"@en.
     }

     DEFINE GPM C ON S1 {
     ?inst pred:InRoom ?r3.
     ?inst pred:name ?n1.
     ?inst pred:status "can use"@en.
    }
 }
```

Query 5: Inventory Management: SPAseq query with Negation Operator

5.7.2. *The case of Optional Operator*

The optional operator selects an event if it matches to the defined GPM expression; otherwise it ignores the event. Since it corresponds to the zero or at-most one occurrence of an event, it suffers from the same issues as discussed for the negation operator. Hence, the remedies for the negation operator can directly be applied to define its semantics. For instance, let us denote the optional operator with '**?**'; then its evaluation can be described using the results of the negation operator as follows:

$$[\![(u,P)?]\!]_{\Sigma}^{[\tau_b,\tau_e]} = [\![(u,P)]\!]_{\Sigma}^{[\tau_b,\tau_e]} \cup [\![(u,P)!\,]\!]_{\Sigma}^{[\tau_b,\tau_e]}$$

The above discussion highlights what kinds of issues arise with the integration of the negation and optional operators in SPAseq and point-outs some of the remedies. In this paper, we do not present the complete semantics of these operator to keep the discussion focused on the core operators of SPAseq. However, in Section 7.7, we present some of the techniques to implement these operators. The processing section presents the qualitative comparative analysis of SPAseq and EP-SPARQL.

6. Qualitative Comparative Analysis

In this section, we present the qualitative comparison between SPAseq and its best competitor EP-SPARQL from the literature. While a complete formal comparison between both is certainly very interesting, we will leave it for future work and focus on a use-case-based comparison of these two languages.

6.1. *Input Data Model*

As discussed in Section 2.2, 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 the 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 observes 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 number of triples in an event are 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 [26]. 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.

6.2. Timepoints Vs Time Intervals

As evident from the data models, the SPAseq temporal semantics is based on points in time, while EP-SPARQL utilises time intervals. The choice of the timepoints for SPAseq is based on the following reasons.

1. The W3C 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]. Their initial recommendations include both timepoints and time-interval based semantics. Furthermore, most of the existing RSP systems employ timepoints-based semantics.
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 [4,8], Zstream [5].
3. Existing CEP and SCEP systems are based on a 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 [34].

The use of timepoints results in a cleaner semantics with the focus on how the RDF graph 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 [35]. 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, [15:30]`), where the two temporal triples can be replaced with a single one with a time-interval (`:person1, :inside-room, :r1, [15:00,15:30]`). 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. an event $(\tau, G) \in \mathcal{S}$ to $([\tau, \tau + k], G)$, while the primitive events – that have no defined end time – can have same start and end timestamps. This would not affect our semantics, since time-interval $[\tau, \tau + k]$ solely covers a single time, namely τ [34].

Considering aforementioned arguments, herein, the semantics of SPAseq are described using timepoints. The description of time-interval based semantics for SPAseq is left for the future work.

6.3. 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 head and shoulders pattern from **UC 2**, where a V-shaped pattern is extended to determine the predictive behaviour of certain values. Such a pattern is extensively used in stock market analysis. However, it does not have a straight forward implication in use cases related to sensor measurements. For instance, consider **UC 1**, where the streams provide sensor information about the power and the weather-related sources. In such a use case, the events will not follow a strict V-shaped pattern, instead multiple events will entail the same values. This requires a relaxed pattern, where multiple events with the same values can be consumed. Figure 3a 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** $e_3.value > e_1.value$); while a relaxed V-shaped pattern is described in Figure 3b. In order to

[2] RDF Stream Abstract Syntax and Semantics: https://goo.gl/It4dtE , last accessed: February, 2017.

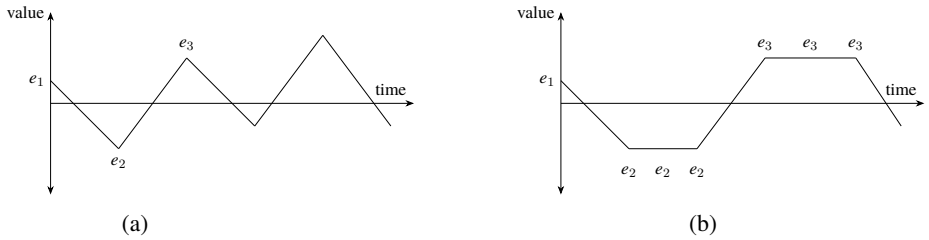

Figure 3. V-Shaped Patterns (a) Without *Kleene+* Operator, and (b) With *Kleene+* Operator

support such a pattern, SPAseq provides the *Kleene+* operator to consume one or more events with the same value. However, EP-SPARQL, and other SCEP languages in the literature, do not provide a *Kleene+* operator and thus only supports the strict V-shaped pattern.

7. Implementing the SPAseq Query Engine

In this section, we move from the theory to the practical implementation of the SPAseq query engine. We first present the NFA$_{scep}$ model that is utilised to compile SPAseq queries, and then provide details regarding the various system's blocks and optimisations used for the SPAseq query engine.

In general, for a CEP language, the set of defined temporal operators are evaluated against the incoming events in a progressive way. That is, before a composite or complex event is detected 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. 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) [36], graph-based representations (such as Event Detection Graphs) [37,5] to merge all the rules within a single structure, and finally Finite State Machine representations, in particular Non-deterministic Finite Automata (NFA) [8,38]. 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 [19], a rule-based engine, mostly focuses on how the complex rules are mapped and executed as Prolog objects and rules, while SASE [4,8] and Zstream [5] focus on query-rewriting, predicate-related optimisations and memory management techniques.

For the efficient evaluation of SPAseq queries, we opt to use an NFA-based execution model. The rational behind it is based on the following points: (i) given the semantic similarity of SPAseq's sequence expressions to regular expressions, NFA would appear to be the natural choice; (ii) NFAs are expressive enough to capture all the complex patterns in SPAseq; (iii) NFAs retain many attractive computational properties of Finite State Automata (FSA) on words, hence, by translating SPAseq queries into NFAs, we can exploit several existing optimisation techniques [8,38].

In addition to the NFA-based model, we use optimisation techniques to evaluate GPM expressions proposed by our system SPECTRA [26]: SPAseq employs an RDF graph model, it not only requires the efficient management of temporal operators, but also the efficient evaluation of graph patterns.

In the following, we first describe the NFA$_{scep}$ model for SPAseq, and later present how SPAseq queries are compiled and evaluated using NFA$_{scep}$. The optimisation techniques for the execution of SPAseq queries are provided in Section 8.

7.1. The NFA$_{scep}$ Model for SPAseq

We extend the standard NFA model [39] in two ways. First, given that SPAseq matches events with GPM expressions, we associate each automaton edge with a predicate, and for an incoming event, this edge is traversed iff the GPM expression is satisfied by this event. Second, in order to handle statefulness between GPM expressions (shared variables), we store in each automaton instance the mappings of those events that have contributed to the state transition of this instance. We call such an automaton model as NFA$_{scep}$ and it is defined as follows:

Definition 18 *An NFA$_{scep}$ automaton is a tuple* $\mathcal{A} = \langle X, E, \Theta, \varphi, x_o, x_f \rangle$*, where*

- *X: a set of states;*

- *E: a set of directed edges connecting states;*
- *Θ: is a set of state-transition predicates, where each* $\theta \in \Theta$, $\theta = (\mathrm{U}, \mathrm{op}, \mathrm{P})$ *is a tuple;* U *is a set of stream names,* op $\in$ { *'&', '+', '|', ';'*} $\cup \{\varnothing\}$ *is a temporal operator, and P is a graph pattern;*
- *φ: is a labelling function* $\varphi : E \rightarrow \Theta$ *that maps each edge to the corresponding state-transition predicate;*
- x_o*:* $x_o \in X$ *is an initial or starting state;*
- x_f*:* $x_f \in X$ *is a final state or acceptation state.*

We define three types of states: *initial* (x_o), *ordinary* (x) and *final* (x_f) states. These state types are analogous to the ones used in the traditional NFA models to implement the operators such as sequence, *Kleene+*, etc. Each state, except the final state, has at least one forward edge. Note that, we use a structure $\mathcal{H} = (\theta_i, \mathcal{A}_i, \Omega)$ to store the set of mappings Ω corresponding to the state-transition predicate θ_i and the automaton instance $\mathcal{A}_i$. Hence, when an event makes an automaton instance traverse an edge, the mappings in that event are properly referenced.

Example 12 *Figure 4 shows the compiled* NFA_{scep} *for the* SPAseq *Query 1 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,* op*, P) consists of three parameters: graph pattern P for the events with stream names (U);* op *describes the type of operator mapped to an edge, for instance edges of state* x_1 *contain the Kleene+ operator. The description of mapping from the* SPAseq *Query 1 to the* NFA_{scep} *in Figure 4 is as follows:*

- *The* SPAseq *Query 1 contains the sequence expression* SEQ(A, B+, C)*, which produces one initial state, two ordinary states and a final state.*

- *State* x_0 *has one edge with state-transition predicate (called as* `GPM A` *in Query 1)* $(u_{s_1}, \varnothing, P_A)$*, where* $U_{s_1} = \{S1\}$ *and* P_A *is the graph pattern. Since the sequence expression in Query 1 only contains the* immediately followed-by *operator, the* NFA_{scep} *can simply transit to the next state on matching the state-transition predicate.*

- *State* x_1 *maps* `GPM B` *with Kleene+. Therefore, it has two edges each with a state-transition predicate* $(U_{s_2}, +, P_B)$*, one with the destination state* x_2*, and other with itself as destination* (x_1) *to consume one or more same kind of events; where* $U_{s_2} = \{S2\}$ *and* P_B *is the graph pattern.*

- *State* x_2 *has one edge, which is used to transit to the next state if an event matches the defined state-transition predicate* $(U_{s_3}, \varnothing, P_C)$*; where* $U_{s_3} = \{S3\}$ *and* P_C *is the graph pattern.*

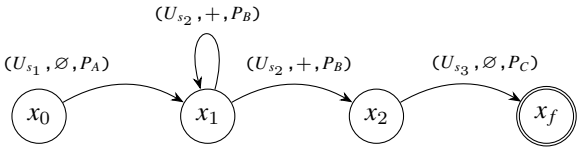

Figure 4. Compiled NFA_{scep} for SPAseq Query 1 with `SEQ(A,B+,C)` expression

State-transition predicates are used to determine the action taken by a state to transit to another. For instance, in Figure 4 the state x_0 transits to x_1, if (1) the incoming event is from the defined stream name U_{s_1}, (2) the evaluation of the graph pattern P_A does not produce an empty set. 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 the *Kleene+* operator, NFA_{scep} will exhibit a non-determinism behaviour, since the state-transition predicates will not be mutually exclusive.

Considering the vocabulary from existing NFA works [8,38], 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 whose final state has reached is a *matched run*, hence denoting that all the defined patterns are matched with the set of selected events. We call the output of the matched run as a *query match*.

7.2. Compilation of SPAseq Queries

As discussed earlier, 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. The compilation process of graph patterns using the traditional relational operators (e.g., selection, projection, cartesian product, join, etc.) within each GPM expression is a straight-forward process from our

earlier work [26]. Herein, we focus on the sequence expression and show how the temporal operators are mapped onto the NFA$_{scep}$.

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, while the window constraints are computed during the evaluation of each automaton run.

Let (u_1, P_1) and (u_2, P_2) be two GPM expressions, the compilation of SPAseq temporal operators onto an NFA$_{scep}$ automaton is described as follows.

– *Simple GPM expression*: The NFA$_{scep}$ for a simple GPM expression with a sequence expression forms a state which transits to the next one with the match of the GPM expression mapped at the state's edge. The NFA$_{scep}$ automaton for a GPM expression (u_1, P_1) is presented in Figure 5.

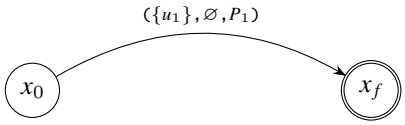

Figure 5. Example of Compilation of the GPM Expression (u_1, P_1)

– *Kleene+*: The *Kleene+* operator selects a set of events if they match to the defined GPM expression. Its automaton is constructed using two edges with one edge having the same source and destination state. Thus, it can detect one or more consecutive events. The corresponding NFA$_{scep}$ for $((u_1, P_1)+)$ is illustrated in Figure 6.

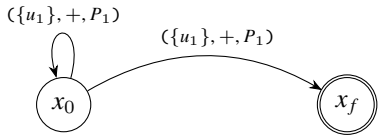

Figure 6. Example of Compilation of the *Kleene+* Operator $((u_1, P_1)+)$

– *Immediately Followed-by*: The construction of NFA$_{scep}$ for this operator is similar to the compilation of a simple GPM expression, 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 7.

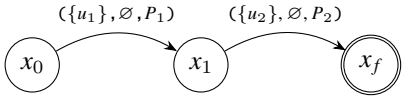

Figure 7. Example of Compilation of the *Immediately followed-by* Operator $((u_1, P_1)\,,\,(u_2, P_2))$

– *Followed-by*: This operator requires the irrelevant events to be skipped. Thus, two different edges emanate from the corresponding state. One has the same source and destination states: this transition matches any kind of event. The second edge is destined for the next state with the defined state-transition predicate. Note that the construction of both flavours of *followed-by* operators (skip-till-next and skip-till-any) is the same with the difference in the evaluation strategies. The corresponding NFA$_{scep}$ automaton for $((u_1, P_1)\,;\,(u_2, P_2))$ is presented in Figure 8, where $U = \{u_1, u_2\}$ is the set of stream names.

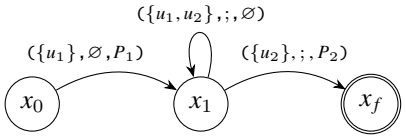

Figure 8. Example of Compilation of the *Followed-by* operator $((u_1, P_1)\,;\,(u_2, P_2))$

– *Conjunction Operator*: This operator detects the simultaneous occurrence of two or more events. Thus, there are two edges for the conjunction state, each destined for the same destination state. The NFA$_{scep}$ automaton for $((u_1, P_1)\&(u_2, P_2))$ is illustrated in Figure 9, where the conjunction state has multiple edges, each having different state-transition predicates.

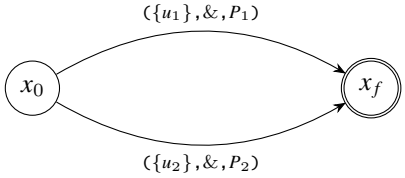

Figure 9. Example of Compilation of the Conjunction Operator $((u_1,P_1)\mathbf{\&}(u_2,P_2))$

– *Disjunction operator*: This operator forms a similar automaton structure as that of conjunction operator, however with the difference of how it is executed for an active run. That is, only one edge has to be matched with the incoming event. The NFA$_{scep}$ automaton for $((u_1,P_1)|(u_2,P_2))$ is illustrated in Figure 10.

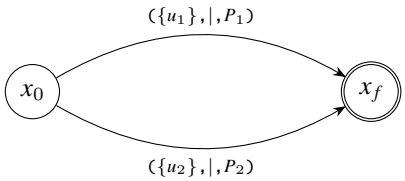

Figure 10. Example of Compilation of the Disjunction Operator $((u_1,P_1)|(u_2,P_2))$

In the aforementioned discussion, we show the compilation of SPAseq operators independently. For the full SPAseq query, these operators can be combined together to form a complex NFA$_{scep}$ automaton. In order to show this, let σ be a sequence with a set of GPM expression and binary/unary operators. The compilation of sequence expression $(\sigma;(u,P)+)$ with the additional *followed-by* and *Kleene+* operators is presented in Figure 11. Notice from Figure 11 how the concatenating process is simply the mapping of the last state of sequence σ onto the initial state of the GPM expression $(u,P)+$.

To conclude, this section presented the mapping of SPAseq queries onto equivalent NFA$_{scep}$ automata. In the next section, we show how NFA$_{scep}$ automata are executed while considering the window constraints defined within a query.

7.3. Evaluation of NFA$_{scep}$ Automaton

The compiled NFA$_{scep}$ automaton represents the model that a matched sequence should follow. Thus,

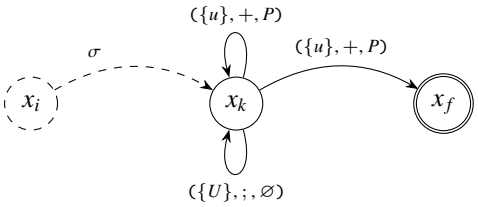

Figure 11. Example of Compilation of the Sequence Expression $(\sigma;(u,P)+)$

Algorithm 1: Processing streamset with NFA$_{scep}$

Input: Σ: streamset, $\mathcal{A}$: NFA$_{scep}$ Automaton, ω: time window

$\mathcal{R} \leftarrow \{\}$: list of active runs
$\mathcal{H} \leftarrow \{\}$: cache history
$\mathcal{D} \leftarrow \{\}$: conjunction edge-timestamp map
foreach *event* $G_e \in \Sigma$ **do**
  get the initial state x_0 from $\mathcal{A}$
  get the final state x_f from $\mathcal{A}$
  get the stream name u from the event G_e
  ProcessEvent $(G_e, u, x_0, x_f, \mathcal{H}, \mathcal{R}, \mathcal{D}, \mathcal{A}, \omega)$

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 a run that reaches to its final state represents a matched sequence.

When a new event enters the NFA$_{scep}$ evaluator, it can result in several actions to be taken by the system. We describe them as follows:

– New runs can be created, or new runs are duplicated (cloned) from the existing ones in order to cater the *Kleene+* operator, thus registering multiple matches.
– If the newly arrived events match to state-transition predicates (θ) of the active states, existing runs transit from one active state to another.
– Existing runs can be deleted, either because the arrival of a new event invalidates 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 ($\mathcal{R}$), (ii) starts a new run or deletes the obsolete ones, (iii) chooses the right event for the state-transition predicates ($\theta \in \Theta$), (iv) calls the GPM evaluator to match

an event with the graph pattern (P) which is provided in the state-transition predicate (θ), and (v) keeps track of the mappings of matched events with a structure $\mathcal{H}$.

Algorithm 2: ProcessEvent with NFA$_{scep}$

Input: G_e: Graph Event, u: stream name, x_0: initial state, x_f: final state, $\mathcal{H}$: cache history, $\mathcal{R}$: list of active runs, $\mathcal{D}$: conjunction edge-timestamp map, $\mathcal{A}$: NFA$_{scep}$ Automaton, ω: time window

1 **Function** ProcessEvent($G_e, u, x_0, x_f, \mathcal{H}, \mathcal{R}, \mathcal{D}, \mathcal{A}, \omega$)
2 **foreach** *run* $r \in \mathcal{R}$ **do**
3 **if** $\omega >$ *initialising time of* r **then**
4 remove r from the list of active runs $\mathcal{R}$
5 **continue**
6 $x_i \leftarrow$ GetActiveState(r)
7 $E \leftarrow$ GetEdgeSet(x_i)
8 get θ for an edge $e \in E$ s.t $\theta \neq \epsilon$ and $\theta = (U_\theta, op_\theta, P_\theta)$
9 **if** $op_\theta =$ '+' **then**
10 KleenePlus ($G_e, u, x_f, \mathcal{H}, \mathcal{R}, r, x_i, E$)
11 **else if** $op_\theta =$ ';' ***or*** $op_\theta =$ '$\varnothing$' **then**
12 EventSelection ($G_e, u, x_f, \mathcal{H}, \mathcal{R}, r, x_i, E$)
13 **else if** $op_\theta =$ '&' **then**
14 Conjunction($G_e, u, x_f, \mathcal{H}, \mathcal{R}, \mathcal{D}, r, x_i, E$)
15 **else if** $op_\theta =$ '|' **then**
16 Disjunction($G_e, u, x_f, \mathcal{H}, \mathcal{R}, r, x_i, E$)
 // Check if an event G_e can create a new run from the initial state x_0
17 $E_0 \leftarrow$ GetEdgeSet(x_0)
18 get θ for the edge $e \in E_0$ s.t $\theta \neq \epsilon$ and $\theta = (U_\theta, op_\theta, P_\theta)$
19 **if** $u \in U_\theta$ ***and*** Gpm (G_e, P_θ, $\mathcal{H}$) **then**
20 initiate a new run r_i of $\mathcal{A}$ with
21 active state x_1
22 $\mathcal{R} \leftarrow \mathcal{R} \cup r_i$

Algorithm 1 presents the initialisation process of various data structures and how a streamset Σ is processed against the NFA$_{scep}$ automaton $\mathcal{A}$. The initialised data structures include (i) a list of currently active runs ($\mathcal{R}$), where each run stores partial matches; (ii) a history cache $\mathcal{H}$ to store the mappings of matched events; (iii) an edge-timestamp map $\mathcal{D}$ to store the mapping of events and their timestamps that are matched at a conjunction operator's state (*lines 1-3*). The algorithm selects the initial state x_0 and final state x_f of automaton $\mathcal{A}$, and the stream name u of the event to be processed (*lines 5-6*). This information along-with the initialised structures is passed to the ProcessEvent function (see Algorithm 2), where each incoming event G_e is matched with the active automaton's runs.

In Algorithm 2, we present the general execution of SPAseq operators with the arrival of an event. The customised execution of SPAseq operators are provided in the proceeding sections. Algorithm 2 begins by iterating over the list of active runs. The list of active runs $\mathcal{R}$ is used to determine if (i) the existing runs are expired or not, i.e. their initialisation time is outside the window boundary, and hence to be deleted (*lines 2-5*); (ii) the active state of the active run can be matched with the newly arrived event (*lines 9-16*). The algorithm starts by comparing the initialising time of the run r under evaluation and the defined time window. The run r is deleted if its outside the defined window (*line 3*). The algorithm then extracts the current active state of the run, and according to the mapped operators it selects the appropriate function for the corresponding operator. In the end, the algorithm checks if the incoming event can start a new run or not (*lines 19-22*). That is, it matches the initial state's (x_0) edge of the automaton $\mathcal{A}$ with the incoming event G_e using the Gpm function (*line 22*) [3]. In case of a match and if the event is from the same stream the edge is waiting for ($u \in U_\theta$), it initiates a new run r_i of the automaton $\mathcal{A}$ with the active state x_1, i.e. it proceeds to the next state (*line 23*). This new run is then added to the list of active runs $\mathcal{R}$ (*line 24*).

The proceeding sections, i.e Sections 7.4, 7.5 and 7.6, explain how the functions defined in Algorithm 2 are implemented to support *Kleene+*, event selection and binary operators of SPAseq respectively.

7.4. Evaluation of the Kleene+ Operator

Previously we show the generic execution for NFA$_{scep}$ automaton for a sequence expression. Herein,

[3] Herein, for sake of brevity, we only show the simple option where the complex operators are not mapped at the initial state. However, in practice, it is implemented using the function for the defined operator at the initial state.

we present how the unary operator, i.e. *Kleene+*, is evaluated. The evaluation of *Kleene+* operator is described in Algorithm 3 with details as follows.

Algorithm 3: Evaluation of the *Kleene+* Operator

```
1  Function KleenePlus(G_e, u, x_f, H, R, r, x_i, E)
2      get θ for an edge e ∈ E s.t θ = (U_θ, op_θ, P_θ)
3      if u ∈ U_θ and GPM (G_e, P_θ, H) then
4          clone a new run r_c from r with active state
             as x_i
5          SetActiveState(r_c, x_{i+1}) where
             x_{i+1} ≠ x_i
           // If it is the final state
6          if x_i = x_f of r_c then
7              a query match has been found
8              remove r from the list of active runs R
9          else
10             R ← R ∪ r_c
11     else if u ∉ U_θ or ¬ GPM (G_e, P_θ, H) then
12         remove r from the list of active runs R
```

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 of the *Kleene+* operator, a new run is duplicated/cloned from the existing one in case of a match. Algorithm 3 presents the evaluation of *Kleene+* operator. It first uses an edge from the active state to employ the comparison of stream names to make sure the event is from the stream the edge is waiting for (*line 3*). The algorithm then uses event G_e, graph pattern P_θ and history cache $\mathcal{H}$ to execute the GPM process (*line 3*). If the newly arrived event G_e is matched with the graph pattern P_θ: (i) a new run r_c is cloned from the run under evaluation with the same active state x_i (*line 4*); (ii) the cloned run transits to the next state x_{i+1} (*line 5*); (iii) if the new active state of the cloned run is the final state then a query match has been found (*lines 6,7*); (iv) otherwise the cloned run is added to the list of active runs $\mathcal{R}$ (*line 9*). It then skips to the next run in $\mathcal{R}$, hence the run under the evaluation stays at the same state. Otherwise, in case of no match, it removes the run r from the list of active runs $\mathcal{R}$ (*lines 10-11*). This way the system can keep track of one or more matched events of the same kind (see Figure 12).

Example 13 *Consider Figure 12. 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 1), 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 $(U_{s_1},\varnothing,P_A)$. Now considering the next event G_e^2 matches $(U_{s_2},+,P_B)$; the automaton results in a non-deterministic move due to the Kleene+ operator at the state x_2. Hence, the algorithm creates a new run r_2 with active state as x_2, i.e. transiting from x_1, while r_1 stays at the same state x_1. When G_e^3 arrives and matches to $(U_{s_3},\varnothing,P_C)$, r_2 moves to the final state and the match for r_2 is complete with events G_e^1, G_e^2 and G_e^3, while r_1 remains active to consume for future events . Finally, after the arrival and match of events G_e^4 and G_e^5 with the corresponding GPM expressions, r_1 reaches the final state with a match using events G_e^1, G_e^4 and G_e^5.*

Algorithm 4: Evaluation of the Event Selection Strategies

```
1  Function EventSelection(G_e, u, x_f, H, R, r,
     x_i, E)
2      get θ for an edge e ∈ E s.t θ = (U_θ, op_θ, P_θ)
3      if u ∈ U_θ and GPM (G_e, P_θ, H) then
4          SetActiveState(r, x_{i+1})
           // If it is the final state
5          if x_i = x_f then
6              a query match has been found
7              remove r from the list of active runs R
8      else if op_θ = ';' and u ∉ U_θ or ¬GPM (G_e,
         P_θ, H) then
9          skip the event for the followed-by operator
10     else
11         remove r from the list of active runs R
```

7.5. Evaluation of the Event Selection Operators

This section presents the evaluation of event selection strategies. Algorithm 4 shows the evaluation of *immediately followed-by* and *followed-by*. Herein, we only discuss the implementation of *followed-by* operator with skip-till-any configuration. The optimised implementation of skip-till-any operator will be the topic of our future endeavours.

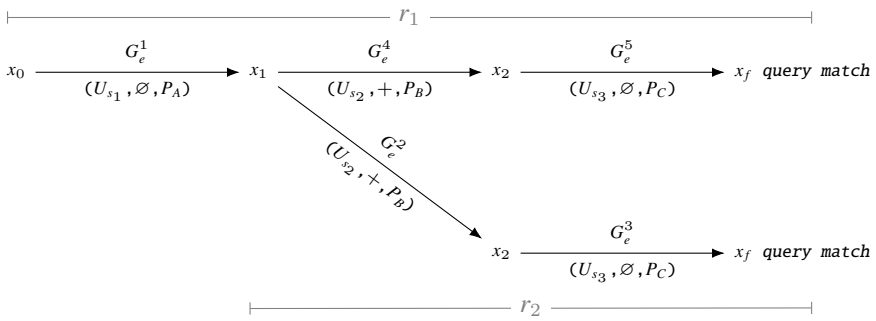

Figure 12. Execution of NFA$_{scep}$ runs for the SPAseq Query 1, as described in Example 13

7.5.1. Immediately Followed-by Operator

The evaluation of the *immediately followed-by* operator is rather simple due to the strictness of how an event should follows other. That is, the incoming event is compared with the state-transition predicate and if there is a match the run transits to the next state; otherwise it is deleted. Therefore, Algorithm 4 first gets the set of edges for the active state and selects the state-transition predicate (*line 2*). It then compares the stream names and the incoming event G_e with the graph pattern P_θ using the Gpm function (*line 3*). If there is a match, the current active state x_i transits to the next state (*line 4*). In case the transited state is the last state a query match is found (*lines 5-6*). Otherwise the run under evaluation is deleted from $\mathcal{R}$ (*line 11*).

7.5.2. Followed-by Operator

From our earlier discussion, the *followed-by* operator (with skip-till-next configuration) skips the irrelevant events utill a query match is completed for the defined sequence expression. This extended functionality is described in Algorithm 4 (*lines 8-9*). That is, it first compares the stream names. If the event is from the same stream, the state's edge is waiting for, i.e. $u \in U_\theta$, it employs the same procedure as described for the *immediately followed-by* operator. In case the arriving event is from a different stream, i.e. $u \notin U_\theta$ (*line 8*) or the event is not matched, the algorithm skips the event (*line 11*). Hence, the run stays at the same state. Algorithm 4 can be extended for the skip-till-any configuration by initiating a new run each time an event is matched with the graph pattern P_θ. However, this will result in a exponential time complexity and will not be suitable for real-world applications.

7.6. Evaluation of the Binary Operators

Finally, we present how the conjunction and disjunction operators are evaluated in an NFA$_{scep}$ automaton.

Algorithm 5: Evaluation of the Conjunction Operator

```
1  Function Conjunction(G_e, u, x_f, H, R, D, r, x_i, E)
2      τ ← GetTimeStamp (G_e)
3      AME ← {} // Already Matched Edges
4      foreach each edge e ∈ E do
5          if (r, x_i, e) ∈ D then
6              AME ← AME ∪ e
7      get timestamp τ_old for a ∈ AME using D
8      if τ ≠ τ_old then
9          remove r from the list of active runs R
10     else
11         E ← E \ AME
12         foreach edge e ∈ E do
13             get θ from the edge e s.t
                θ = (U_θ, op_θ, P_θ)
14             if u ∈ U_θ and Gpm (G_e, P_θ, H) then
15                 insert (r, x_i, e) and τ in D
16                 AME = AME ∪ e
17         E ← GetEdgeSet(x_i)
18         if |AME| = |E| then
19             SetActiveState(r, x_{i+1})
               // If it is the final state
20             if x_i = x_f then
21                 a query match has been found
22                 remove r from the list of active
                    runs R
```

Algorithms 5 and 6 show the execution of these operators.

7.6.1. 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 the consecutive events having the same timestamp. This means we need to track the edges of a conjunction state that are already been matched and their timestamps. Therefore, we use a mapping structure ($\mathcal{D}$) to store the mapping between a tuple (r, x, e) and a timestamp τ of the matched events; where r is the run, x is state with conjunction operator and e is the edge of the state that is matched with an event having timestamp τ. The evaluation of the conjunction operator is presented in Algorithm 5. Its main evaluation starts by: (i) obtaining the timestamp τ of the newly arrived event (*line 2*) and (ii) initialising a set (AME) to store already matched edges (*line 3*). It then iterates over the set of edges E and checks if any of the edges have already been matched or not, using the mapping structure $\mathcal{D}$, while adding the matched edges to the AME set (*lines 4-6*). It then extracts the timestamp τ_{old} from an element in AME set using the mapping structure D (*line 7*). This timestamp τ_{old} is used to check if the newly arrived event has the same timestamp τ. Otherwise, the algorithm removes the run r from $\mathcal{R}$, since it has violated the condition of conjunction operator (*lines 8-9*). If $\tau = \tau_{old}$, it extracts the set of already matched edges (AME). It then removes the edges from edge set E that are already matched with the previous events (*line 11*). This pruned set E is then used to match the incoming event with the selected edges. The algorithm iterates over E and for each $e \in E$ it takes the state-transition predicate θ to compare the stream name and graph pattern using the GPM function (*line 12-14*). If there is a match it marks the edge as matched by adding its mapping in $\mathcal{D}$ and inserting the edge in the AME set (*lines 15-16*). In the end, the algorithm again extracts all the edges for the conjunction state and uses the AME set to check if all the edges are matched or not (*line 17-18*). In case all the edges have completed the matching procedure it transits the run r to the next state (*line 19*). If the transited state is the last state a query match is found for the conjunction operator.

7.6.2. Disjunction Operator

The disjunction operator resembles with the conjunction operator. However, in this case the run can transits to the next state if the incoming event matches to at-least one of the state's edges. Algorithm 6 presents

Algorithm 6: Evaluation of the Disjunction Operator

```
1  Function DISJUNCTION(G_e, u, x_f, H, R, r, x_i, E)
2      Matched ← false
3      foreach edge e ∈ E do
4          get θ from the edge e s.t θ = (U_θ, op_θ, P_θ)
5          if u ∈ U_θ and GPM (G_e, P_θ, H) then
6              Matched ← true
7      if Matched then
8          SETACTIVESTATE(r, x_{i+1})
           // If it is the final state
9          if x_i = x_f then
10             a query match has been found
11             remove r from the list of active runs R
12     else
           // If none of the edges e ∈ E
              match with the event G_e
13         remove r from the list of active runs R
```

the evaluation of the disjunction operator. It starts by initiating a variable Matched to keep track if any of the state's edge is matched with the incoming event. It then iterates over each edge using its state-transition predicate θ (*line 3-10*), and compares the stream names U_θ and the graph pattern P_θ with the event G_e (*line 5*). If any of the edge matches to the event, it updates the value of the Matched variable (*line 6*). Next if the value of Matched variable is *true*, it transits the run to the next state (*line 8*). Similarly to the other algorithms, if such transition resulted in arriving at the final state of the automaton, a query match is found and the run is deleted (*line 9-10*). Otherwise, if none of the edges are matched with the event, it deletes the run under evaluation (*line 13*).

In the previous sections, we presented the execution of the SPASEQ temporal operators and how they use the NFA_{scep} automaton to implement the required functionalities. The discussion regarding the run-time optimisations is provided in Section 8.

7.7. Compilation and Evaluation of Negation and Optional Operators

In Section 5.7, we discussed the case of integrating negation and optional operators in SPASEQ. We described the issues that can arise while integrating their semantics with the core SPASEQ operators. Herein, we

showcase the techniques to compile and implement these operators. To allow the expressivity of optional (**'?'**) and negation (**'!'**) operators, we extend op and φ in NFA$_{scep}$ definination (Definition 18), such that $op \in \{$ '**&**', '**+**', '**|**', '**;**', '**?**, '**!**"$\} \cup \{\varnothing\}$ and $\varphi : E \rightarrow \Theta \cup \{\epsilon\}$, where ϵ denotes the instantaneous transition [40]. The compilation process of optional and negation operators is described as follows:

– *Optional*: The optional operator selects an event if it matches to the defined GPM expression; otherwise it ignores the event. Its compilation results in two edges: one with an ϵ-transition and the other with the defined state-transition predicate. The corresponding NFA$_{scep}$ automaton for $((u_1, P_1)?)$ is illustrated in Figure 13. The ϵ transition allows the transition to the next state without a match.

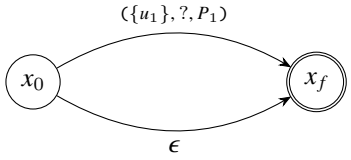

Figure 13. Compilation of the Optional Operator for $((u_1, P_1)?)$

– *Negation*: This operator detects if either no match of an event occurs or there is no occurrence of the expected event. 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 $((u_1, P_1)!)$ is illustrated in Figure 14.

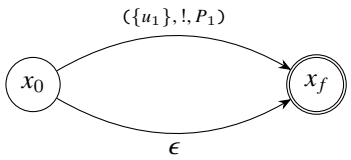

Figure 14. Compilation of the Negation Operator for $((u_1, P_1)!)$

We now present the evaluation of the negation and optional operators. The evaluation functions of these operators can be integrated in Algorithm 2.

Algorithm 7: Optional Operator's Evaluation

```
1  Function OPTIONAL(G_e, u, x_f, H, R, r, x_i, E)
2      get θ for an edge e ∈ E s.t θ ≠ ε and
         θ = (U_θ, op_θ, P_θ)
3      if u ∈ U_θ then
4          if GPM (G_e, P_θ, H) then
5              SETACTIVESTATE(r, x_{i+1})
6          else
7              if PROCEEDINGSTATE(G_e, u, x_i, x_f)
                 then
8                  SETACTIVESTATE(r, x_{i+2})
9              else
10                 SETACTIVESTATE(r, x_{i+1})
11     else if u ∉ U_θ then
12         if PROCEEDINGSTATE(G_e, u, x_i, x_f) then
13             SETACTIVESTATE(r, x_{i+2})
14         else
15             SETACTIVESTATE(r, x_{i+1})
       // If it is the final state
16     if x_i = x_f then
17         a query match has been found
18         remove r from the list of active runs R
```

7.7.1. Optional Operator

The main difference between the evaluation of the optional and negation operators is to differentiate between the occurrence and non-occurrence of an event. For instance, given a sequence expression SEQ(A, B?, C) and events G_e^1 and G_e^2. If G_e^1 matches A then we have to match G_e^2 with both B and C; since it is possible that the event G_e^2 does not match with B but with C, and in such a case we can have a query match. Therefore, the evaluation of the optional operator makes sure that the event that does not match with the optional state is compared with the next state's state-transition predicate. The evaluation of the optional operator is shown in Algorithm 7. It first employs the edge with state-transition predicate $\theta \neq \epsilon$ (*lines 1*) and uses the stream names to make sure the event is from the stream the edge is waiting for (*line 3*). The algorithm then uses event G_e, graph pattern P_θ and the history cache $\mathcal{H}$ to execute the GPM process (*line 4*). If the event G_e matches with the selected edge, it transits to the next state (*line 5*). Otherwise it checks the event G_e with the next state-transition's predicate (*line 7-10*) using the CHECKNEXTSTATE function in Algorithm 8. If the event G_e matches to the next state-transition pred-

icate – considering if it is not the final state – it transits the active state x_i to x_{i+2} (*line 8*). Furthermore, if the event is not from the desired stream, the run takes the ϵ-transition and compares the next state-transition predicate to either transit to the next state x_{i+1} or next of the next state x_{i+2} (*lines 12-15*). During the evaluation of the operator, if the current state reaches the final state, the algorithm output the query match (*line 16-18*).

Algorithm 8: Comparing Proceeding State's Transition Predicate

```
1  Function ProceedingState(G_e, u, x_f, H, R, r)
2      select proceeding state x_{i+1}
3      if x_{i+1} = x_f then
4          return false
5      E ← GetEdgeSet(x_{i+1})
6      get θ for an edge e ∈ E s.t θ ≠ ϵ and
         θ = (U_θ, sf_θ, op_θ, P_θ)
7      if u ∈ U_θ and Gpm (G_e, P_θ, H) then
8          return true
9      else
10         return false
```

7.7.2. Negation Operator

The negation operator is evaluated in a similar fashion compared with the optional operator. However, in this case, the run transits to the next state (producing an identity element) if there is no match with the mapped graph pattern P_θ. The evaluation of the negation operator is described in Algorithm 9. It begins by selecting the state-transition predicate and compares the stream names and graph pattern P_θ with the event G_e using Gpm function (*lines 4*). If Gpm returns *false*, i.e. the event is not matched with the graph pattern, the algorithm uses the CheckNextState function to determine if such an event can be matched with the next state-transition predicate x_{i+1} (*line 5*), as described for the optional operator. In case the current active state x_i is matched with the event G_e, it means the run has violated the negation operator and should be deleted (*line 10-11*). If the event is from a different stream, the evaluation of the negation operator is same as that of optional operator (*lines 12-19*).

Algorithm 9: Negation Operator's Evaluation

```
1  Function Negation(G_e, u, x_f, H, R, r, x_i, E)
2      get θ for an edge e ∈ E s.t θ ≠ ϵ and
         θ = (U_θ, sf_θ, op_θ, P_θ)
3      if u ∈ U_θ then
4          if ¬Gpm (G_e, P_θ, H) then
5              if ProceedingState(G_e, u, x_i, x_f)
                 then
6                  SetActiveState(r, x_{i+2})
7              else
8                  SetActiveState(r, x_{i+1})
9          else
10             remove r from the list of active runs R
11             return
12     else if u ∉ U_θ then
13         if ProceedingState(G_e, u, x_i, x_f) then
14             SetActiveState(r, x_{i+2})
15         else
16             SetActiveState(r, x_{i+1})
       // If it is the final state
17     if x_i = x_f then
18         a query match has been found
19         remove r from the list of active runs R
```

7.8. Design of the SPAseq Query Engine

Having provided the details of compiling the SPAseq queries onto NFA$_{scep}$ automata and their evaluation strategies, herein we provide an overview of the system architecture of SPAseq query engine.

Figure 15 shows the architecture of the SPAseq query engine. Its main components are *input manager*, *queue manager*, *query optimiser*, *NFA evaluator*, and an RDF graph processor in *GPM evaluator*. In the following, we briefly discuss these components.

The queue and input managers do their usual job of feeding the required data into the NFA evaluator. Since our system employs streamsets, there are multiple buffers to queue the data from a set of streams. Moreover, the newly constructed events for SPAseq queries can also be fed back to the queue manager for further processing. The incoming data from streams are first mapped to the numeric IDs using dictionary encoding[4].

[4] Dictionary encoding is a usual process employed by a variety of RDF-based systems [41,42]. It reduces the memory footprints by replacing strings with short numeric IDs, and also increases the system performance by using numeric comparisons instead of costly string comparisons.

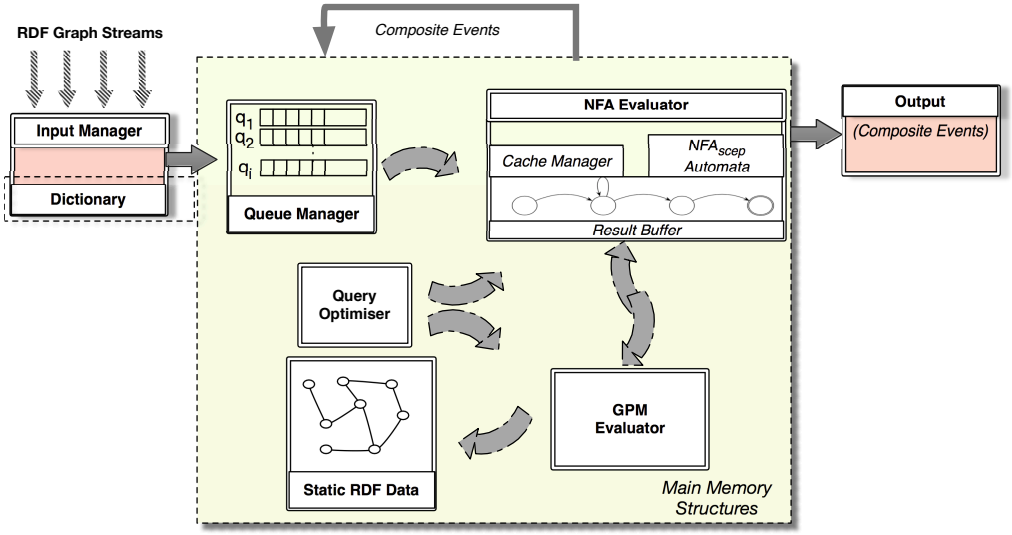

Figure 15. Architecture of the SPAseq Query Engine

The input manager also utilises an efficient parser[5,6] to parse the RDF formatted data into the internal format of the system.

At the heart of SPAseq query engine, the GPM evaluator uses the GPM expressions, events, cache history and edge-timestamp mappings, as described in Algorithms 3, 4, 5 and 6 to match the defined graph pattern with the incoming events. We employ our SPECTRA GPM engine [26] for this task. SPECTRA is a main memory graph pattern processor but uses specialised data structures and indexing techniques suitable for the optimised evaluation of RDF graph events. We employ two main operators of SPECTRA for SPAseq and they are described as follow:

- The SummaryGraph operator enables the system to avoid storing all the triples within an event by exploiting the structure of the graph patterns. That is, the graph pattern P is used to prune all the triples within an event that do not match the subjects, predicates or objects defined in P. The pruned set of triples within an event, called *Summary Graph*, are materialised into a set of vertically partitioned tables called *views*. Each view, a two-column table (s, o) stores all the triples for each unique predicate in a summarised event.
- The QueryProcessor operator employs the set of views to implement multi-join operations using incremental indexing. That is, for each view a *sibling list* is used to incrementally determine the set of joined subject/objects that belong to a specific branch within a graph. This results in an incremental solution, where the creation and maintenance of the indexes are the by-product of the join executions between views, not of updates within a defined window.

The use of specialised indexing techniques and data structures enables SPECTRA to outperform existing RSP systems up to an order of magnitude [26]. Thus, employing such optimisations also aids the evaluation of GPM expressions in SPAseq. Furthermore, the evaluated events can also be enriched using the static RDF data, where such data are loaded into the memory using the vertically partitioned tables.

Next comes the NFA evaluator of SPAseq. It contains the compiled NFA_{scep} automata and employs the GPM evaluator to compute the GPM expressions mapped on the state-transition predicates. Its subcomponent, the cache manager stores the mappings of matched events and mappings for the conjunction operator, i.e. cache history ($\mathcal{H}$) and conjunction edge-timestamp map ($\mathcal{D}$). Finally, the query optimiser employs various techniques to reduce the load for GPM evaluator and the number of active runs. The detailed

[5] We employ the performance intensive NxParser, which is a non-validating parser for the Nx format, where x = Triples, Quads, or any other number.

[6] NxParser: https://github.com/nxparser/nxparser, last accessed: June, 2017.

description of query optimiser is presented in the proceeding section.

8. 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 7, many different strategies have been proposed to find an optimal way of utilising CPU and memory in CEP systems. 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 RDF graph streams. In this section, we describe how such techniques are applicable for SCEP, and also propose new ones considering the query processing over a streamset. First, we review the evaluation complexity for the main operators of the SPAseq query language.

8.1. 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 transit to the next one. Given n number of events in a streamset, the cost of comparing a graph pattern P with an n events for unary operators and event selection strategies is described as follows:

$$cost_n = \sum_{i=1}^{n} \mathbf{c}(P, G_e^i),$$

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 1 in [43]. Now we extend this cost to include the cost of the binary operators (Bop). Given n events and k *Bop* operators, the cost function is extended as follows:

$$cost_n^k = \sum_{i=1}^{n} \sum_{j=1}^{k} \mathbf{c}(P_j, G_e^i),$$

Notice that due to the k number of *Bop* operator, each event in worst case scenario has to be matched with all the defined GPM expressions for the *Bop* operators.

Prior works analysing the complexity of NFA evaluation often consider the number of runs created by an operator and employ upper bounds on its expected value [8,38]. We adopt the same approach for analysing the complexity of NFA_{scep} evaluation. Indeed, 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. Query operators that result in creating new runs or those which 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 a GPM expression over each event.
2. We ignore the selectivity measures of the state-transition predicates, i.e. the events that are not matched, either skipped or result in deleting a run of an NFA_{scep} automaton. Hence, our focus is 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*, *Kleene+*, conjunction and disjunction.

Theorem 1 *The upper bound of evaluation complexity of event selection strategies, i.e.* immediately followed-by, followed-by, *is linear-time* $\mathcal{O}(n)$*, where n is the total number of runs generated for the n input events.*

Proof Sketch. None of the operators described in Theorem 1 duplicate runs from the existing ones. 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 and for n number of events there can be only n runs. □

Although the upper bound of the operators described in Theorem 1 has the same evaluation complexity, there exists discrepancies when considering the real-world scenarios. Let us consider the case of event selection operators: the *followed-by* operator skips irrelevant events, while *immediately 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 expected duration of a run is shorter for the *immediately followed-by* operator, since a run 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).

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: (i) more than one edge has 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 of a conjunction/disjunction operator, and n input events, we can provide the upper bound on the complexity of these operators as follows:

Theorem 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 edges of a state.*

Proof Sketch. For the conjunction and disjunction operators no new runs are initiated from old ones. If an event arrives at active state x, 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 the conjunction operator). However, for both operators, each incoming event can end up traversing the whole set of k mapped edges. Thus, even if the number of active runs remains the same, each event may have to be matched with a number of GPM expressions mapped at the state's edges. For k such edges and n events, in worst case the cost will be $\mathcal{O}(n \cdot k)$. □

Theorem 3 *The upper bound of evaluation complexity of each Kleene+ operator is quadratic-time $\mathcal{O}(n^2)$ for 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 events 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 Theorems 1 and 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.

8.2. 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. 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 are actually affected by an event.

8.2.1. Pushing the Temporal Window

As mentioned earlier, the window defined in a SPASEQ query constraints the matches to be executed over the unbounded streams. It is thus desirable to evict the runs that contain events outside the window boundaries 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, pushing the evaluation of the temporal window at the top of the processing stack, deletes the runs without first evaluating the state-transition predicates. It consequently decreases the size of the active runs list. In Algorithm 2 (*lines 3-5*), we push the window check before iterating over the active runs list.

8.2.2. Pushing the Stateful Predicates

Stateful predicates define the joins amoung a set of graph patterns. These joins can be defined through the `FILTER` expression or within the set of graph patterns (see SPASEQ Query 1). With the arrival of an event, there are two 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 issue an expensive GPM process, and if the stateful joins are not fruitful, GPM process would be of waste since 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 pushes the stateful joins as early as possible in the processing stack.

As an example of this, consider the SPASEQ Query 1. 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.

8.2.3. Indexing Runs by Stream Names

SPASEQ queries are evaluated over a streamset, where the edges from each state contain the stream name which is used to match the graph patterns. Therefore, each active state waits for a specific type of event

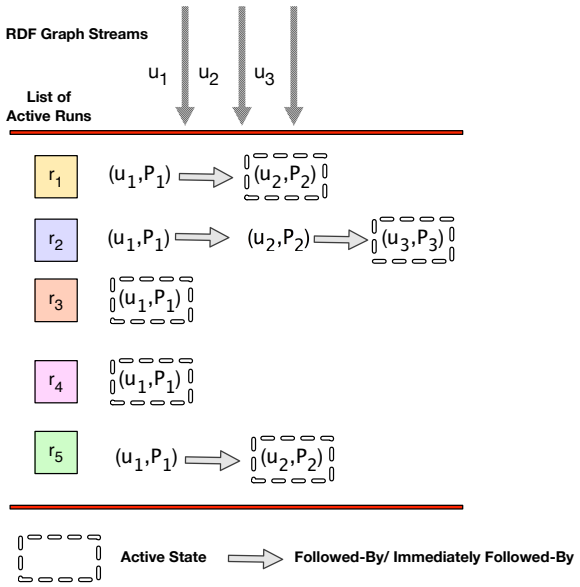

Figure 16. Processing Streamset over Active Runs

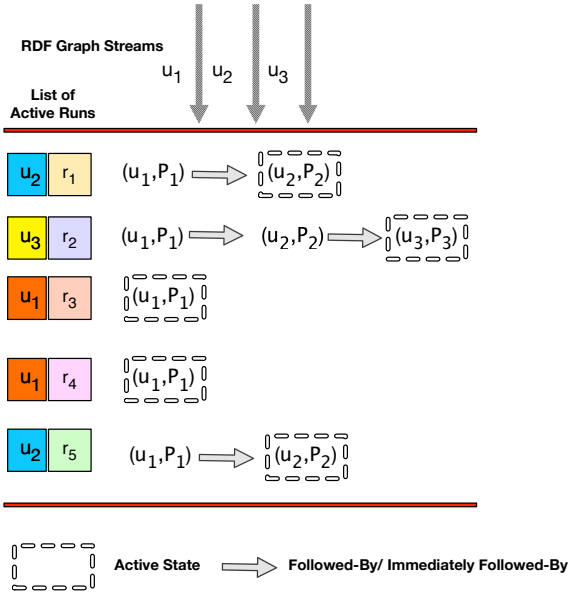

Figure 17. Partitioning Runs by Stream Names

from a specific stream, and later invokes the GPM evaluator. We illustrate it through an example.

Example 14 *Figure 16 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 name u_2, the active run r_2 is waiting for an event from stream with name u_3, and active runs r_3 and r_4 are waiting*

for the events from the stream with name u_1. According to Algorithm 2, for each incoming event from the set of streams, we have to iterate over all the active runs and then match the stream name and GPM expressions respectively.

Indexing runs by stream names allows 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 generally lower. Thus, we index each run by the stream name of its active state (see Figure 17). More precisely, the index takes the stream name 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. These indices 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.

8.2.4. Memory Management

Although in-memory data access is extremely fast compared to 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 to avoid the unnecessary memory utilisation. The three main data structures that require tweaking are the *cache manager*, the *result buffer* and the *indexed active run list*. 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 expired runs to be deleted. These runs are added to a pool: when a new runs is created, we try to recycle the memory from such pool. This limits the initialisation of new runs 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 runs can be found in theoretically constant time.

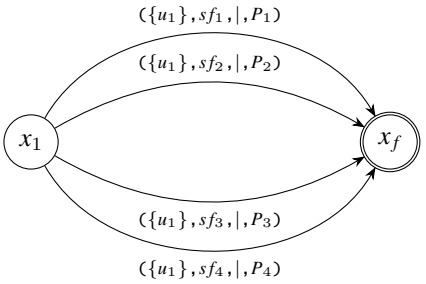

Figure 18. Compilation of Disjunction Operator for $((u_1, P_1) \mid (u_1, P_2) \mid (u_1, P_3) \mid (u_1, P_4))$

8.3. 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. 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 18 shows the related NFA_{scep} automaton. Now consider an input stream u_1, and an event G_e^i at time τ_i arrives at the state x_1. The basic sequential way of processing G_e^i would be to first match with P_1 then P_2, P_3 and finally with P_4: since all the graph patterns are waiting for an event from the stream u_1. Now the question is how to choose the less costly graph pattern to be selected first by the state, such that if it matches the automaton moves to the next state, hence complying to the optional operator.

The optimal way of processing the disjunction 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 GRAPH-SUMMARY operator be handy for them (see Section 7.8).
2. Adaptively gather the statistics about the cost of matching a specific graph pattern, and sort the graph pattern accordingly.

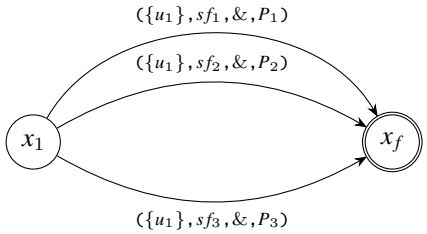

Figure 19. Compilation of Conjunction Operator for $((u_1, P_1)$ & (u_1, P_3) & $(u_1, P_3))$

Let us focus on the first approach. As according to Theorem 1 in [44], 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 a 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 reduce the cost of the GPM operation. Following this reasoning, we keep a sorted set of graph patterns $\langle P_1, P_2, \ldots, P_k \rangle$ within the state which mapped the conjunction/disjunction operator, each associated with a stream name 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 [26]. The state can utilise this set to first inspect the less costly graph patterns for the incoming events. This can lead to a less costly disjunction 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 19 shows the NFA$_{scep}$ automaton for it. Hence, for the state x_1 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_1, P_2) respectively. Then the automaton waits for an event to satisfy GPM expression (u_1, 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 state that maps conjunction operator.
2. After the conformity of the above constraint, choose graph patterns according to their costs (as discussed for disjunction operator).

The main idea underlying our lazy evaluation strategy is to avoid unnecessary high cost of the 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 work [5] on buffering the events and processing them as batches.

Algorithm 10 shows the lazy evaluation of the conjunction operator and it extends Algorithm 5. It employs an event buffer $\mathcal{B}$ to cache the set of events having the same timestamps. The algorithm first takes the set of edges E for an active state and timestamp τ of the newly arrived event (*lines 1-2*). It then checks if there exists any previously buffered event in $\mathcal{B}$. If so it checks their timestamp τ_{old} and the timestamp τ of the newly arrived event (*line 5*). If the timestamps do not match it deletes the run under evaluation while removing all the previously buffered events in $\mathcal{B}$ (*lines 4-7*). Otherwise it adds the event G_e in the event buffer $\mathcal{B}$, which is later used to evaluate the unmatched edges (*line 10*). At this stage, the algorithm prunes the edge set E with the edges that have already been matched – using the mapping structure $\mathcal{D}$ (*line 11*) (as described in Algorithm 5). If the number of unmatched edges in E is equal to the number of buffered event $\mathcal{B}$, it starts

Algorithm 10: Optimised evaluation of the conjunction operator

Input: G_e: Graph Event, u: stream name, r: active run, x_f: final state, r: active run, $\mathcal{H}$: cache history, $\mathcal{D}$: conjunction edge-timestamp map, $\mathcal{B}$: event buffer

```
1  x_i ← GetActiveState(r)
2  E ← GetEdgeSet(x_i)
3  τ ← GetTimeStamp (G_e)
4  if |B| ≠ ∅ then
5      get τ_old from an event G_e^k ∈ B
6      if τ ≠ τ_old then
7          B ← ∅
8          remove r from the list of active runs R
9      else
10         B = B ∪ G_e
11         remove edges from E using D that are
             already matched
12         if |E| = |B| then
13             sort E according to the graph patterns
14             foreach each edge e ∈ E do
15                 get θ from the edge e
16                 foreach each event G_e^i ∈ B do
17                     if u ∈ U_θ and
                         GPM(G_e^i, P_θ, sf_θ, H) then
18                         remove G_e^i from event
                             buffer B
19                         insert (r, x, e) and τ_i in D
20             E ← GetEdgeSet(x_i)
21             get the set of matched edges AME
                 from D
22             if |E| = |AME| then
23                 SetActiveState(r, x_{i+1})
                   // If it is the final
                      state
24                 if x_i = x_f then
25                     a query match has been found
```

the matching process using the lazy evaluation strategy. Before starting the matching process, it first sorts the edges according to the selectivity of the graph patterns in state-transition predicates, hence using the low cost edges first (*line 13*). The algorithm then iterates over the sorted edges E and the buffered events set to match the selected edge (*lines 12-22*). If an edge $e \in E$ matches the buffered event $G_e^i \in \mathcal{B}$, its mapping is added into $\mathcal{D}$ and the matched event is removed from the buffer $\mathcal{B}$ (*lines 18-19*). In the end, the algorithm has to determine if all the edges of the active states are matched and should it transit to the next state or not. Therefore, it examines the number of actual edges of the state and the number of them that have already been matched (*line 25*). If all the edges are matched the run transits to the next state. Otherwise, it waits at the same state to receive more events having the same timestamps (*line 28*).

In this section, we presented various strategies to optimise the evaluation of SPASEQ temporal operators. In the next section, we present the quantitative analysis of the SPASEQ operators and the effect of our optimisation strategies.

9. 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 setup, and later we analyse the system performance in the form of questions. SPASEQ is implemented in Java. To support reproducibility of experiments, it is released under an open source license[7].

9.1. 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 [45,46]: it is based on the fractal time series and provides properties such as randomness, non-determinism, chaos theory, etc. The SMD data generator is openly available at Github[8]. We generated a dataset of more than 20 million triples and 10 million RDF graph events.

[7] SPASEQ: https://github.com/Gillani0/spaseq, last accessed: June, 2017.

[8] Stock Market Dataset: https://github.com/spaseq/stock.data.gen, last accessed: June, 2017.

The UMass Smart Home Dataset (SHD) [47] 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 for renewable energy sources. We use a smart grid ontology [48] to map the raw eventual data into N-Triples format for three different streams: the *power stream* ($\mathcal{S}_1$), the *power storage* stream ($\mathcal{S}_2$) and the *weather stream* ($\mathcal{S}_3$). In total the dataset contains around 30 million triples, 8 million events.

Queries. We use two main queries for the above mentioned datasets: **UC 1** (Query 1) and **UC 2** (Query 2). That is, a smart grid pattern and a heads and shoulder pattern. Both queries are further extended for various experiments.

Constraints. The execution time/throughput of the evaluated 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 Java Virtual Machine (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. These configuration enables us to test both the worst case performance and the real-world expected performance.

- *Config. RG*: Random Generation (RG) of events. For this configuration, we randomly ordered the generated events in the streams, which means some events follow the defined pattern in the evaluated queries and some events do not. Hence, simulating the real-world characteristics.
- *Config. SG*: Sequence-based Generation (SG) of events. For this configuration, we ordered the generated events according to the defined patterns in the evaluated queries. This results in a 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 8u112.

9.2. 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. each other?

First we describe the setup for this set of experiments. In order to compare the relative complexity of the operators, we employ the SMD dataset, extensions of **UC 1** (Query 1), and *Config. SG* to generate streams. It allows to make sure that each operator in question has the exact number of matches. For this set of experiments, we use the *immediately followed-by* operator between unary operators.

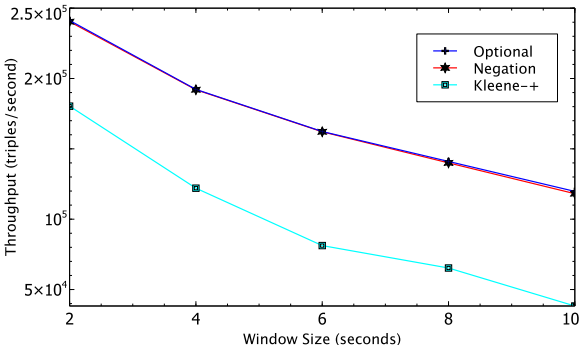

Figure 20. Performance Measures of Optional, Negation and *Kleene+* Operators

Figure 20 shows the results of our experiments. As expected, the negation and optional operators show linear scaling behaviour with the increase in the window size. That is, the number of runs generated for each of these operators is relatively proportional to the number of events that result in starting a new run from its initial state x_0. 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 an extra cost for the *Kleene+* operator to process an event.

The negation and the optional operators share the same cost and execution strategies, hence, they result in the similar throughput measures (Figure 20). Note that for the both of these operators, if an event does not match to the negation/optional state's state-transition predicate, it is again used to check if it can match with the next state's state-transition predicate. Thus it results in an extra evaluation of a GPM expression compared with simple sequence expression. The evaluation of the aforementioned set of experiments showcases the result parallel to the one discussed for the evaluation complexity of SPAseq operators (Section 8.1).

Question 2. How do the binary operators conjunction and disjunction perform w.r.t. each other?

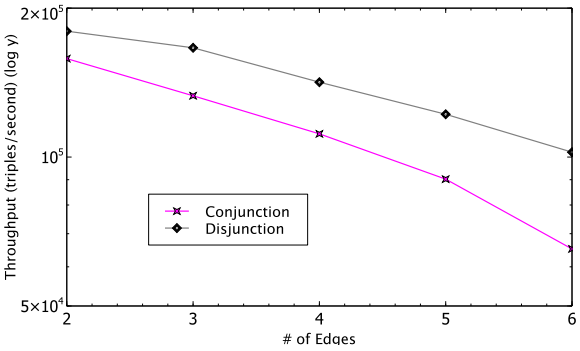

Figure 21. Comparison of *Conjunction* and *Disjunction* Operators

For this set of experiments, we again use the SMD dataset, **UC 2** queries, and *Config. SG* to generate streams. Since all the events emanate from a single stream, it results in the most complex case: all the GPM expressions mapped at the conjunction/disjunction state's edges expect the events with the same stream name, hence, 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 21 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. 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 21. As expected, the conjunction operator scales linearly with the increase in number of edges to be matched, while the disjunction operator scales sub-linearly, since it has to match only one of the edges to transit to the next state. Moreover, not surprisingly, the conjunction operator is much more expensive than the disjunction operator. The conjunction operator has to match the complete set of GPM expressions mapped on the set of edges, while the disjunction operator results in few matches and its corresponding state transits to the next one as soon as there is one match with either of them. Note that, for this set of experiments, we use the lazy evaluation strategy for the conjunction operator, its 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. each other?

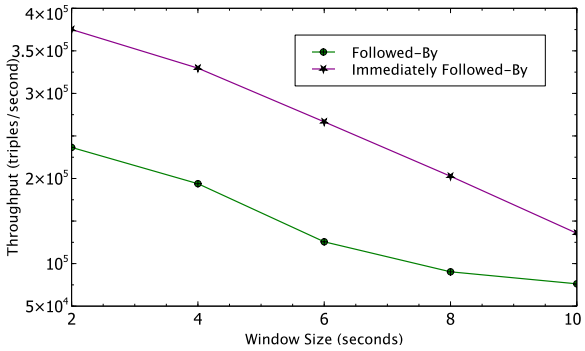

Figure 22. Comparison of *Followed-by* and *Immediately Followed-By* Operators

For this set of experiments, we use the SHD dataset and **UC 1** (Query 1), since the effect of the operators in question is quite distinguishable in a multi-stream environment. We tested both operators using only *Config. RG*, since *Config. SG* produces events according to the defined patterns, hence no events are skipped for the *followed-by* operator and both operators would have the similar performance measures. Figure 22 shows the evaluation of these operators using *Config. RG*. As we can see the *immediately followed-by* operator is less expensive compared with the *followed-by* operator: the *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 deletes the run as soon as it violates its defined constraint. The runs for the *followed-by* operator are deleted if the timestamps of the selected events for the run are outside the defined window. 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 larger list of runs to be matched.

Effects of Optimisation Strategies

Question 4. How does the strategy for indexing runs by stream names affect the performance of the system?

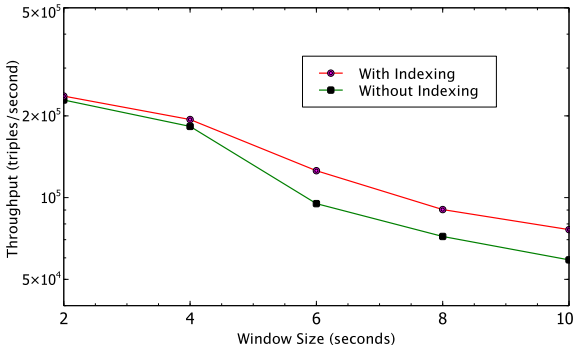

Figure 23. Analysis of Indexing Runs by Stream Names

In order to determine the effectiveness of indexing runs by stream names, we employ the SHD and **UC 1** query with *followed-by* operator, and *Config. RG*; since the *followed-by* operator is costly compared with the *immediately followed-by* operator. Recall from Section 8.2, we index runs by stream name, thus when an event arrives, only the runs whose active state is waiting for such an event are used. Consequently, it reduces the overhead of going through the whole list of available active runs. Figure 23 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 numbers of runs are produced/remain 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 is produced with a smaller set of them waiting for an event from a specific stream.

Question 5. How does the lazy evaluation affect the performance of the conjunction operator?

For this set of experiments, we again employ the SMD dataset, extension of **UC 2** with conjunction operator containing 4 edges, and *Config. RG* to generates events: *Config. SG* produces events according to the defined pattern and the effects of the lazy evaluation would not be obvious in such case. Figure 24 shows the results of the conjunction operator with lazy and

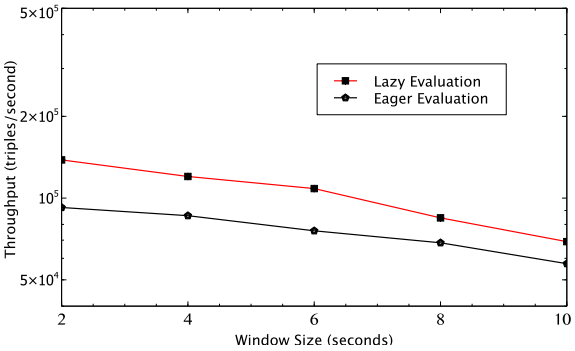

Figure 24. Lazy vs Eager Evaluation of Conjunction Operator

eager evaluation strategies. Recall from Section 8.3, 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 24, the lazy evaluation performs much better on smaller windows and relatively better on larger ones: the eager evaluation results in a larger number of useless calls to the GPM evaluator, while lazy evaluation performs a batch-based call to the GPM evaluator. Thus, with lazy evaluation, a set of events is evaluated against a set of GPM expressions, only if all the buffered events (for a conjunction state) has the same timestamp. For the smaller window, if the number of buffered events is not equal to the number of edges from a conjunction state, the GPM evaluator is not invoked. Hence, with the expiration of the window, the runs are deleted without matching events and without using the additional resources. Contrary to this, the eager evaluation strategy calls the GPM evaluator for each incoming event and a large 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. 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 [49].

For the dataset and queries, we used the SMD dataset, its respective query for the V-shaped pattern (see **UC** 2) and *Config. SG* 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 (making it W ot head and should pattern) with a fixed window size of 8 seconds. Note that, since we used *Config. SG* for this set of experiments (sequence-based event generation), both *followed-by* and *immediately followed-by* operators would have the same performance measures. Furthermore, we used a simple V-shaped pattern query since EP-SPARQL does not support the *Kleene+* operator.

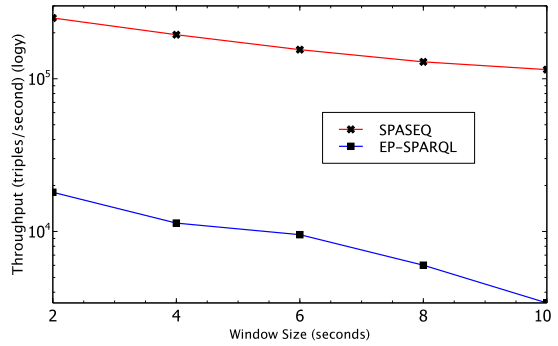

Figure 25. Comparative Analysis of SPAseq and EP-SPARQL over Variable Window Size

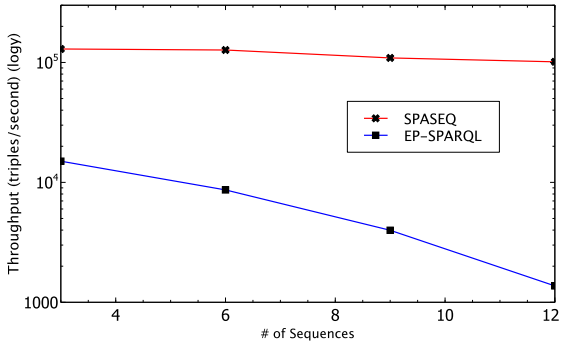

Figure 26. Comparative Analysis of SPAseq and EP-SPARQL over Variable # of Sequences

Figures 25 and 26 show the performance of both systems. In Figure 25, both systems are evaluated by varying the size of the window in seconds, while in Figure 26, we evaluate both systems by varying the size of the pattern to be matched, i.e. the number of sequences. From these results, we can see that the SPAseq yields much higher throughput compared with EP-SPARQL for both scenarios. One simple reason for this is as follows: the results provided in [28] show that RSP systems such as CQELS outperforms ETALIS, while as shown in [26] that SPECTRA outperforms other RSP system. Based on this, sicce ETALIS is the underlying engine of EP-SPARQL and SPECTRA of SPAseq, the performance of the GPM would be supirioir for SPAseq. Furthremore, the complexity introduced by the temporal operators in SPAseq is well managed by our NFA_{scep} model and optimisation techniques compared with EP-SPARQL.

From Figure 25, the performance of EP-SPARQL degrades quadratically with the increase of the window size: 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 the 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 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 (see Figure 26): with the increase in sequence operators, the active life of each run also increases, however, employing the above mentioned optimisation techniques greatly reduces the life-span and the number of active runs.

10. Conclusion

In this paper, we presented the syntax, semantics and implementation of a SCEP query language called SPAseq. We provided the motivation behind SPAseq and pointed out various qualitative differences between other SCEP languages. Such analysis showcased the usability and expressivity of the SPAseq query lan-

guage. 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. We also proposed an NFA_{scep} model to map the SPAseq operators and showed how they can be evaluated over a streamset. Furthermore, we also provided multiple optimisation techniques to evaluate SPAseq operators in an optimised manner. Lastly, while utilising real-world and synthetic datasets we showcased the usability and performance of SPAseq query engine. Our future endeavours include: extension of the language with new operators, a through semantic comparative analysis with the EP-SPARQL, further optimisation strategies for the *Kleene+* operator and the evaluation of the SPAseq operators in a distributed environment. We believe that SPAseq can ignite the SCEP research community and will open the doors for the new insights and optimisation techniques in this field.

References

[1] Lukasz Golab and M. Tamer Özsu. Issues in data stream management. *SIGMOD Rec.*, 32(2):5–14, June 2003.

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

[3] Gianpaolo Cugola and Alessandro Margara. Processing flows of information: From data stream to complex event processing. *ACM Comput. Surv.*, 44(3):15:1–15:62, June 2012.

[4] 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, pages 407–418, New York, NY, USA, 2006. ACM.

[5] 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, pages 193–206, New York, NY, USA, 2009. ACM.

[6] David Luckham. The power of events: An introduction to complex event processing in distributed enterprise systems. In *Rule Representation, Interchange and Reasoning on the Web*. 2008.

[7] Barzan Mozafari and Zeng. High-performance complex event processing over xml streams. In *SIGMOD*, 2012.

[8] Jagrati Agrawal, Yanlei Diao, Daniel Gyllstrom, and Neil Immerman. Efficient pattern matching over event streams. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, pages 147–160, New York, NY, USA, 2008. ACM.

[9] Barzan Mozafari, Kai Zeng, Loris D'antoni, and Carlo Zaniolo. High-performance complex event processing over hierarchical data. *ACM Trans. Database Syst.*, 38(4):21:1–21:39, December 2013.

[10] Lars Brenna, Alan Demers, Johannes Gehrke, Mingsheng Hong, Joel Ossher, Biswanath Panda, Mirek Riedewald, Mohit Thatte, and Walker White. Cayuga: A high-performance event processing engine. In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*, SIGMOD '07, pages 1100–1102, New York, NY, USA, 2007. ACM.

[11] Thomas BERNHARDT and Alexandre VASSEUR. ESPER-complex event processing,. In *Online Article*, 2010.

[12] Drool fusion. http://www.drools.org/. Accessed: 2016-06-03.

[13] Danh Le-Phuoc, Minh Dao-Tran, Josiane Xavier Parreira, and Manfred Hauswirth. A native and adaptive approach for unified processing of linked streams and linked data. In *ISWC*, pages 370–388. 2011.

[14] Davide Francesco Barbieri and Braga. C-SPARQL: Sparql for continuous querying. In *WWW*, pages 1061–1062, 2009.

[15] Jean-Paul Calbimonte, Oscar Corcho, and Alasdair J. G. Gray. Enabling ontology-based access to streaming data sources. In *ISWC*, pages 96–111, 2010.

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

[17] Özgür L. Özcep Veronika Thost, Jan Holste. On Implementing Temporal Query Answering in DL-Lite (extended abstract). In Diego Calvanese and Boris Konev, editors, *Proceedings of the 28th International Workshop on Description Logics, Athens,Greece, June 7-10, 2015.*, volume 1350 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2015.

[18] Daniele Dell'Aglio, Minh Dao-Tran, Jean-Paul Calbimonte, Danh Le Phuoc, and Emanuele Della Valle. A query model to capture event pattern matching in RDF stream processing query languages. In *Knowledge Engineering and Knowledge Management - 20th International Conference, EKAW 2016, Bologna, Italy, November 19-23, 2016, Proceedings*, pages 145–162, 2016.

[19] Darko Anicic, Sebastian Rudolph, Paul Fodor, and Nenad Stojanovic. Stream reasoning and complex event processing in etalis. *Semant. web*, 3(4):397–407, October 2012.

[20] Yanlei Diao and Neil Immerman. Sase+: An agile language for kleene closure over event streams. *UMASS Technical Report*, 2007.

[21] Jagrati Agrawal and Yanlei Diao. Efficient pattern matching over event streams. In *SIGMOD*, 2008.

[22] Raman Adaikkalavan and Sharma Chakravarthy. Snoopib: Interval-based event specification and detection for active databases. *Data Knowl. Eng.*, 59(1):139–165, October 2006.

[23] Alessandro Margara, Jacopo Urbani, Frank van Harmelen, and Henri Bal. Streaming the web: Reasoning over dynamic data. *Web Semantics: Science, Services and Agents on the World Wide Web*, 25(0):24 – 44, 2014.

[24] Srdjan Komazec, Davide Cerri, and Dieter Fensel. Sparkwave: Continuous schema-enhanced pattern matching over RDF data streams. In *DEBS*, pages 58–68, 2012.

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

[26] 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, pages 17:1–17:12, New York, NY, USA, 2016. ACM.

[27] Minh Dao-Tran and Danh Le Phuoc. Towards enriching CQELS with complex event processing and path navigation. In *Proceedings of the 1st Workshop on High-Level Declarative Stream Processing co-located with the 38th German AI conference (KI 2015), Dresden, Germany, September 22, 2015.*, pages 2–14, 2015.

[28] Danh Le-Phuoc, Minh Dao-Tran, Josiane Xavier Parreira, and Manfred Hauswirth. A native and adaptive approach for unified processing of linked streams and linked data. In *Proceedings of the 10th International Conference on The Semantic Web - Volume Part I*, ISWC'11, pages 370–388, Berlin, Heidelberg, 2011. Springer-Verlag.

[29] Davide Francesco Barbieri, Daniele Braga, Stefano Ceri, and Michael Grossniklaus. An execution environment for c-sparql queries. In *Proceedings of the 13th International Conference on Extending Database Technology*, EDBT '10, pages 441–452, New York, NY, USA, 2010. ACM.

[30] Özgür Lütfü Özçep, Ralf Möller, and Christian Neuenstadt. A stream-temporal query language for ontology based data access. In *KI 2014: Advances in Artificial Intelligence - 37th Annual German Conference on AI, Stuttgart, Germany, September 22-26, 2014. Proceedings*, pages 183–194, 2014.

[31] Richard Cyganiak, David Wood, and Markus Lanthaler. RDF 1.1 Concepts and Abstract Syntax. Technical report, W3C, January 2014.

[32] Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez. Semantics and complexity of sparql. *ACM Trans. Database Syst.*, 34(3):16:1–16:45, September 2009.

[33] P. H. Kevin Chang Carol L. Osler. Head and shoulders: Not just a flaky pattern. In *FRB of New York Staff Report No. 4*, 1995.

[34] Jayant R. Haritsa and T. M. Vijayaraman, editors. *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.

[35] 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, pages 180–191, San Francisco, CA, USA, 1996. Morgan Kaufmann Publishers Inc.

[36] Charles L. Forgy. Expert systems. chapter Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem, pages 324–341. IEEE Computer Society Press, Los Alamitos, CA, USA, 1990.

[37] 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*, pages 2–9, Feb 1994.

[38] 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, pages 217–228, New York, NY, USA, 2014. ACM.

[39] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

[40] Ken Thompson. Programming techniques: Regular expression search algorithm. *Commun. ACM*, 11(6):419–422, June 1968.

[41] Thomas Neumann and Gerhard Weikum. The rdf-3x engine for scalable management of rdf data. *The VLDB Journal*, 19(1):91–113, February 2010.

[42] Jeremy J. Carroll, Ian Dickinson, Chris Dollin, Dave Reynolds, Andy Seaborne, and Kevin Wilkinson. 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, pages 74–83, New York, NY, USA, 2004. ACM.

[43] Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez. Semantics and complexity of sparql. *ACM Trans. Database Syst.*, 34(3):16:1–16:45, September 2009.

[44] 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, pages 214–225, New York, NY, USA, 2016. ACM.

[45] Bruce M. Kapron, Valerie King, and Ben Mountjoy. Dynamic graph connectivity in polylogarithmic worst case time. In *SODA*, pages 1131–1142, 2013.

[46] Natcha Simsiri, Kanat Tangwongsan, Srikanta Tirthapura, and Kun-Lung Wu. Work-efficient parallel and incremental graph connectivity. In *CoRR*, volume abs/1602.05232, 2016.

[47] David Wood, Markus Lanthaler, and Richard Cyganiak. RDF 1.1 concepts and abstract syntax. In *W3C Recommendation, Technical Report*, 2014.

[48] Syed Gillani, Frederique Laforest, Gauthier Picard, et al. A generic ontology for prosumer-oriented smart grid. In *EDBT/ICDT Workshops*, pages 134–139, 2014.

[49] Mark Sanderson and Justin Zobel. Information retrieval system evaluation: Effort, sensitivity, and reliability. In *Proceedings of the 28th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '05, pages 162–169, New York, NY, USA, 2005. ACM.