University of Zurich[UZH]

Jörg-Uwe Kietz
Thomas Scharrenbach
Lorenz Fischer
M K Nguyen
Abraham Bernstein

TEF-SPARQL: The DDIS query-language for time annotated event and fact Triple-Streams

TECHNICAL REPORT – No. IFI-2013.07

2013

University of Zurich
Department of Informatics (IFI)
Binzmühlestrasse 14, CH-8050 Zürich, Switzerland

ifi

Jörg-Uwe Kietz, Thomas Scharrenbach, Lorenz Fischer, M K Nguyen, Abraham Bernstein
TEF-SPARQL: The DDIS query-language for time annotated event and fact Triple-Streams
Technical Report No. IFI-2013.07
Dynamic and Distributed Information Systems
Department of Informatics (IFI)
University of Zurich
Binzmuehlestrasse 14, CH-8050 Zurich, Switzerland
http://www.ifi.uzh.ch/ddis/

University of Zurich[UZH]

Jörg-Uwe Kietz
Thomas Scharrenbach
Lorenz Fischer
Minh Khoa Nguyen
Abraham Bernstein

TEF-SPARQL: The DDIS query-language for time annotated event and fact Triple-Streams

TECHNICAL REPORT – No. IFI-2013.07

October 2013

University of Zurich
Department of Informatics (IFI)
Binzmühlestrasse 14, CH-8050 Zürich, Switzerland

ifi

J.-U. Kietz et al.: TEF-SPARQL: The DDIS query-language for Triple-Streams
Technical Report No. IFI-2013.07, October 2013
Dynamic and Distributed Information Systems Group (DDIS)
Department of Informatics (IFI)
University of Zurich
Binzmühlestrasse 14, CH-8050 Zürich, Switzerland
URL: http://www.ifi.uzh.ch/ddis.html

Contents

Chapter 1

Introduction

The amount of data available on the Web (e.g., on the Semantic Web and Linked Open Data Cloud (LOD)) is growing at an astounding speed. An increasing number of these data-sources are dynamic (i.e. their content changes over time) or even represent continually updating phenomena (such as the stock exchange, sensor networks, social networks, or the continuous arrival of intelligence data). In many cases they also contain some kind of temporal information - either explicitly given through temporal constraints (e.g. limiting the validity of a statement by giving a start date and some live span) or implicitly (i.e. defining its validity from the moment it is made available until it is superseded by an update).
The data model underlying relational databases has severe limitations when integrating data from several heterogeneous sources and requiring clear semantics [Stonebraker and Cetintemel, 2005]. Thus, graph based data models like RDF have emerged in the last decade. They are nowadays accepted as an industry standard and are vital components of the Semantic Web and the Web of Data [Bizer et al., 2009]. Real-time processing of linked data on Web scale is no longer possible offline but requires stream-processing systems that guarantee to correctly answer queries within acceptable time-frames whilst (i) continuously consuming triples and (ii) utilizing a limited amount of computational resources (i.e. space and time).
Traditional approaches (see Chapter 2) address these requirements by defining a (time- or space-) window over which the queries are evaluated: newly arriving triples evict the oldest ones from the window of consideration. This approach allows for continuous updates since a triple consumption process can continuously update the window (often defined as a "logically circular" region of memory) and a query process can evaluate queries over the window. It also clearly limits memory consumption via the size of the window. The time-window approach is, however, severely limited as it (1) cannot cope with queries that will only match in time-spans longer than the chosen window, (2) cannot evaluate queries with temporal constraints as all triples within the window are usually regarded as being true without any temporal constraint, and (3) does not account for possibly-varying validity-spans of different parts of a query (just consider a query that involves both schema- and instance-related triple-patterns, where the former can be assumed to be long-living and the latter can change continuously). (4) does not account for event delimited windows, i.e. episodes or frames
In this report we first give a brief overview of related work (chapter 2) before we motivate and illustrate the main design decision of the language (chapter 3). We then formally specify syntax and semantic of our language and give some basic properties of our language (chapter 4). We then illustrate on several use-cases how the language can be used to solve typical problems in stream-processing (chapter 5) and conclude with a short discussion (chapter 6).

Chapter 2

Related Work

The following sections are organized as follows: We first describe the characteristics of queries over streams, how they differ from traditional queries, and what types of operators are necessary to express them. Since most research in stream processing has been conducted using relational data models, we are then going to list research on relational stream querying systems. After that, we review systems that, while still processing streams, do this on a stream of triples as opposed to a stream of table rows.
Since, the field of stream processing is vast, we are only going to list the work that is most closely related to ours here and point to other survey papers on event processing [Owens, 2007] and complex event processing (CEP) [Lajos et al., 2010] for more a more extensive overview of the field.

2.1 Temporal Operators

The main difference between static data and streams of data is the temporal aspect. Traditional query languages such as SQL or SPARQL treat the database as a static snapshot. Even though the contents of the database might change over time, the change itself is transparent to the user and the query. Active databases provide some primitive ways of reacting to data mutation using triggers. Triggers allow a user to specify a condition that has to be met for a piece of code to be executed. They do not take temporal relationships of records into account, however. Similarly, one can use a conventional query language to filter a stream of tuples. More complex queries that take the temporal relationships into account, require temporal language constructs. These temporal operators can assigned to one of two categories: Operators that define a *range* of data in the stream (1a) and operators that describe the *temporal order* of data records (1b). A data range is often defined as a window or a frame that moves over the stream and it is often used to compute some aggregate value over the data records it contains. Temporal relationships are mostly used in the event processing (EP) field, where one is interested in a combination of events that have specific temporal relationships. Some stream processing systems only support one or the other, while others support both types. For the queries we want to find answers to, we need a system that supports both types.

2.2 Queries over Streams of Relational Data

Most stream processing systems that exist today operate on traditional relational data. The STREAM system by Arasu et al. [2004] is a general purpose data stream management system (DSMS). Streams are modeled as relations, into which records are inserted or deleted from. Its *CQL* language [Arasu et al., 2005] is based on SQL and allows continuous queries over *relational* data streams using windows. The size of the window is defined

by either a number of rows or a time value on a per-stream basis. There is no support for temporal ordering operators. Similarly to triggers, the existence of a record can be made dependent on the occurrence of other records in the stream using the IStream and DStream operators. The technology of the STREAM system was commercialized in the Coral8 CEP engine which through several acquisitions is part of SAP's Sybase since early 2010.
Another DSMS is TelegraphCQ [Chandrasekaran et al., 2003], which allows a user to register continuous queries (CQ) over static and streamed relational data. The query language of TelegraphCQ is also SQL based. Similarly to STREAM, TelegraphCQ lets a user define the size of the windows and the slide by which a windows is moved on a per-stream basis. Also there is no support for operators to define temporal relationships between events. The system has been commercialized by Truviso, which in turn has been acquired by Cisco Systems in Spring 2012.
Cayuga [Brenna et al., 2007, Demers et al., 2007, 2006], *Sase* [Wu et al., 2006], *Sase+* [Diao et al., 2007], *CEDR* [Barga et al., 2007] and *ZStream* [Mei and Madden, 2009] also work with *relational* data. In contrast to the systems described above, they are event processing systems. As such, they focus on the temporal relations of the data tuples (events) in the stream.
The Cayuga Event Language (CEL) is based on SQL. Its NEXT operator allows the user to define a sequence, in which events have to occur in the stream. The FOLD operator works similarly to the Kleene+ operator. However, in addition to a regular Kleene+ operator the FOLD operator also supports an aggregation expression, which allows the user to create aggregate values over a range defined by the data, as opposed to a date range. This is similar to the concept of *Frames* described in [Maier et al., 2012], where the range of the window is defined by data, rather than a fixed size. A query can use the output of another query, which essentially allows the user to create nested queries.
The strength of the *Sase+* language is the SEQ operator and its variants. They allow the user to specify an order in which events have to or must not occur in the stream. The WITHIN operators allows for the definition of a time based window, which is defined for the whole query. The slide size of the window cannot be defined. *Sase+* also supports aggregate functions such as max, min, avg, count, and sum.
The *CEDR* query language supports event sequencing operators such as SEQUENCE, ALL, ANY, ATLEAST, ATMOST. These operators filter a stream for a sequence or the occurrence of events in a given time frame. In contrast to other languages *CEDR* also supports an array of negation operators.

2.3 Queries over Triple Streams

All of the above systems process streams of *relational* data. In contrast to this, our system processes streams of triples using a query language that builds on the SPARQL query language. Several temporal extensions to the SPARQL have been proposed in literature. The ones most closely related to our work, are listed below.
Continuous SPARQL (C-SPARQL) is an extension to SPARQL that allows the user to query a stream of triples [Barbieri et al., 2009]. Using a special FROM clause the user defines the size and the slide (step) of a window on a per-stream basis. These windows can then be queried like conventional SPARQL endpoints. The system supports aggregates, grouping, and a special timestamp function, which allows for temporal comparison of triples. In [Barbieri et al., 2010] the authors present an implementation of the system, using Stanford's *STREAM* system for the dynamic event processing and Sesame [1] as the RDF datastore for storing and retrieving persistent data. In [Hoeksema and Kotoulas,

[1] http://www.openrdf.org

2011] Hoeksema et al. present work in progress of implementing C-SPARQL on distributed stream processing framework S4 [2].
In their paper, Anicic et al. [2011] present EP-SPARQL. They propose several powerful operators to SPARQL that allow the user to query for sequences of triples arriving in a stream. In ETALIS [3] the authors provide a Prolog implementation of EP-SPARQL which next to classic operators such as conjunction, disjunction and negation also supports aggregates, sequences and all of Allen's temporal operators.
SPARQL-ST [Perry and Sheth, 2009] is a spacio-temporal extension to SPARQL that allows for filters over temporal and geographical attributes of triples. In contrast to C-SPARQL, EP-SPARQL, and our solution, this system is not capable of running continuous queries over a stream of data, but adds temporal filters for queries against an otherwise mostly static datastore.

[2]http://incubator.apache.org/s4
[3]http://code.google.com/p/etalis

Chapter 3

Motivation of Design Decision

3.1 Episode-based matching

Most Stream processing systems are based on the notions of (sliding) time-windows that define the the relevance of incoming data for the scope of queries. The oldest systems had a fixed window defining which data have to be kept in a DB-Cache to answer queries. Newer system have multiple windows each specified as part of the queries to process. While this effectively reduces the infinite amount of incoming data to a (mostly) manageable size, it has the serious limitation that the window size cannot depend on the data, but must be specified in advance at system start or query formulation/deployment.
In a recent paper, [Maier et al., 2012] argues that it is important that stream processing systems are able to detect episodes on streams, detect them accurately, and detect them promptly. They presented several scenarios, in which content-based frames are better suited for solving the problem than fixed-size windows: "We see at least three issues with using windows to identify episodes:

- Window sizes are fixed in terms of tuples or time. Reducing the window size to obtain better accuracy on episode boundaries leads to an increased number of windows.
- Windows are continually produced, meaning that we continue to generate windows (and other results that must be processed) during uninteresting periods.
- Windows are often incorporated into aggregate operators to produce summary values over the windows. We would like the flexibility combine episodes detected on one stream with data from another stream."[Maier et al., 2012, p.1-2]

They therefore defined Frames as a partitioning of the stream into (sub-)sequences of events, for which

- a specified data attribute exceeds a specified threshold
- a minimal duration is exceeded
- and which are maximal (no frame is included in another), drained (i.e. minimal amount of frames) and complete (cover all possible candidate frames).

While this is an interesting and important step towards episode based processing, it lacks a full integration of the frame definition into the matching process.
Complex Event processing systems don't have a sliding window, but most of them require the specification of a max time (space) based attention span per query. This defines a kind of dynamic window per query, events contributing to partial matches are kept as long as they are in the attention span and discarded afterwards. Cayuga [Demers et al.,

2007] offers a FOLD construct to specify a data-segment based on the content of the data, but this is limited to compute aggregations and does not allow pattern matching on the specified segment. A pure data or event delimited attention is to our knowledge not used so far[1].

	max time/max space	data driven
Stream Processing, streamed data are buffered in a DB and processed by (normal DB) queries on them	Sliding Window: data are buffered by a sliding window of specified size or time	Frame, the data are partitioned based on simple properties of the data
Complex Event Processing	partial matches expire after a fixed amount of time or space	partial matches expire when some event happens or fact is no longer valid

Figure 3.1: Stream vs Event Processing with limited or data determined episodes

3.2 Integration of Events and temporarily Facts

In our system we distinguish between, Events, i.e. things that happen(ed) and Facts, i.e. things that are true for a specified amount of time. Peter bought a book is an event that happened at a specific point in time. It never becomes invalid that this happened, but it may be irrelevant after a while, i.e. when it is to old to influence any active query. Someone owns a book is a fact, and this fact is usually only true for a restricted period of time. It becomes true after he bought the book or when someone gifted it to him. If nothing else happens it stays true, in which case it belongs to the current state. But there are also other events that can occur, which would terminate (and negate it for the future) the fact that he owns the book, such as when he looses it or when he gives it to a friend. We aren't aware of any other system that differentiates and integrates time-dependent facts and events. Some [Barbieri et al., 2010] systems integrated events and immutable facts, and some (ab-)use events to represent time-depended factual knowledge, but no one integrated time-dependent facts into event-processing so far.
But we think this it is important in an event processing system to have time dependent facts, as

- events start or finish validity of temporal facts, and

[1]This has of course the danger of never discarded nor completed partial matches. We plan to handle them with intelligent garbage collection, i.e. purge the most unlikely completable ones when memory is needed.

- the validity of facts influence the interpretation of events.

Suppose we get (basic) events about birth, relocation and dying[2], e.g.:

```
JohannSebastianBach bornIn Eisenach [1685]
JohannSebastianBach relocatedTo Ohrdruf [1695]
JohannSebastianBach relocatedTo Lueneburg [1700]
JohannSebastianBach relocatedTo Weimar [1703]
JohannSebastianBach relocatedTo Arnstadt [1703]
JohannSebastianBach relocatedTo Muehlhausen [1707]
JohannSebastianBach relocatedTo Weimar [1708]
JohannSebastianBach relocatedTo Koethen [1717]
JohannSebastianBach relocatedTo Leipzig [1723]
JohannSebastianBach diedIn Leipzig [1750]
```

These events would allow us to infer where a person lived when, e.g.

```
JohannSebastianBach liveIn Eisenach [1685,1695]
...
JohannSebastianBach liveIn Leipzig [1723,1750]
```

But so far only the ETALIS [Anicic et al., 2010] event processing system allows us to deploy the rules to make such inferences. In our language this can be expressed as following:

```
CONSTRUCT ?Person liveIn ?Place
     WHERE (SINCE ?Person bornIn ?Place)
          UNION
          (REPLACE ?Person liveIn ?OldPlace
               ON ?Person relocatedTo ?Place,
                  ?Place =/= OldPlace)
          UNION
          (TILL (?Person diedIn ?SomeWhere
                 HAPPENSWHILE ?Person liveIn ?Place))
```

Before 1685 this query cannot infer anything about JohannSebastianBach. Starting with his birth the query infers JohannSebastianBach liveIn Eisenach [1685,∞] till his first relocation that finishies it to JohannSebastianBach liveIn Eisenach [1685,1695] and newly infers JohannSebastianBach liveIn Ohrdruf [1695,∞] and so on for all further relocations, till his dead event that changes JohannSebastianBach liveIn Leipzig [1723,∞] to JohannSebastianBach liveIn Leipzig [1723,1750].
This illustrates the effects of events on the computation and validity of facts. But also the validity of facts have an influence on the interpretation of events. Let's come back to our book buying example. We could define that a person buys a book (a complex event) in an online shop by pressing the "BuyButton" (basic event), if and only if the book is currently in the shopping basket (a temporal fact) and the account balance (of the selected payment method) of the person contains enough money to pay (another temporal fact) the book current price (a third temporal fact). Together with this buying event (an output event) an event to charge this persons account should be triggered. In our language this would look as following.

```
CONSTRUCT ?Person buys ?Book, ?Person accountCharge ?Price WHERE
    (?Person presses "BuyButton")
    DURING
    (?Person hasInShoppingBasket ?Book,
```

[2]This is not really a time critical-example for a stream-engine, but as soon as you replace, birth, relocated and dying by entering, observedAt and leaving an area or building you have one.

```
?Book hasPrice ?Price,
?Person accountCovers ?Price)
```

3.3 Orientation towards the Semantic Web

The earlymost stream processing languages offspring from the database community and the query languages where oriented on data base standards, i.e. on SQL. However, we think that the Semantic Web is nowadays the environment into which stream-processing has to be integrated. The Semantic Sensor Web integrates millions of sensors, millions of and computers generate events in interaction with the Semantic Web (publishing and subscribing to news, making contacts, buy and sell goods and information, ...) and the Web of Data provides an enormous amount of background knowledge useful to interpret the events.
Therefore, we base our language for defining temporal event and fact patterns on the W3C recommendation SPARQL [Harris and Seaborne, 2012]. It defines a protocol and a language for querying RDF graphs. As could be shown, the expressive power of SPARQL queries is equal to the of SQL [Angles and Gutierrez, 2008]. In contrast to SQL that relies on a relational model for data, RDF and consequently SPARQL offer a number of advantages over the relational model [Stonebraker and Cetintemel, 2005]:
In RDF, both schema information as well as instances are expressed facilitating the same syntax. Hence, we can query for schema information. This is particularly useful when yet unknown links between two instances shall be discovered. Assume, for example, we described an instance of a terrorist $?x$ and and an instance of a suspect to terror with $?y$. Opposed to SQL we can query for potential links explicitly using SPARQL:

```
SELECT ?x, ?p, ?y
WHERE {
  {SUBQUERY FOR ?x}
  {SUBQUERY FOR ?y}
  ?x ?p ?y .
  {CONSTRAINTS ON ?p}
}
```

The result of that query not only returns the potential links between terrorist and suspects, but allows further constraining the relationships between these. Note that we can define all kinds of transitive patterns for matching not only first-degree links between $?x$ and $?y$ but also nearly arbitrary indirect ones (e.g. links of at least length 3, etc.). Note that SPARQL cannot work on the Kleene closure.
Besides the simple data model of RDF gravely simplifies the integration of new datasets, and there exist various tools for turning relational data and many more formats into RDF. Finally, there exist very sophisticated definitions for schema languages, the most popular of which are the W3C recommendations RDFS (RDF Schema), OWL (Web Ontology Language), SKOS (Simple Knowledge Organization System), and RIF (Rule Interchange Format). These advanced schemata allow for various ways of constraining and describing the actual data. More important, for languages, such as RDFS and OWL, there exist formal semantics that allow for sound logic inference.
As a result, RDF and SPARQL are the perfect candidates for integrating various different data sets not known ex-ante. Patterns can be defined in a most flexible way and reasoning support can be added, if necessary.

Chapter 4

Language Reference

4.1 Time Annotated Triples and Streams

A time instant τ is a value taken from a set of discrete equidistant values,[1] i.e. positive natural numbers $\mathcal{N} = 1, 2, \ldots$ or one of the three special time instants $\{0, \eta, \infty\}$. 0 is interpreted as "at system start" or "since ever" η is interpreted as "currently" or "up to now" and ∞ is interpreted as "forever". For any $\tau \in \mathcal{N} : 0 < \tau < \eta < \infty$. $now() \rightarrow \mathcal{N}$ is a monotonically non-decreasing function returning the current point in time.[2]
A time interval T_i is a pair of time instants $[\tau_s, \tau_e]$ for which it holds that $\tau_s \leq \tau_e$. A time stamp T_s is a pair of equal time instants $[\tau, \tau]$. $T = T_i \cup T_s$ is a time annotation. For time annotations the function $duration(T) \rightarrow xs : dayTimeDuration \cup \{\infty\}$ computes the time interval length. See [Malhotra et al., 2010] for the specification of xs:dayTimeDuration and functions working on it and xs:dateTime.
A basic triple t is a tuple $\langle s, p, o \rangle \in I \times I \times (I \cup L)$ consisting of subject s, predicate p, and object o, as defined in [Klyne and Carroll., 2004] where $U = (I \cup L)$ denotes the set of possible terms.[3] We assume the predicates P be partitioned into three disjoint sets P_f for facts and P_e for events and P_s for special predicates like $rdf : type$.
A time annotated triple $e = \langle s, p, o \rangle [\tau_s, \tau_e]$ with $p \in P_e$ and $0 < \tau_s \leq \tau_e < \infty$ is called event triple (or just event), $\mathcal{E}$ is the set of all event triples. We say e is produced at $[\tau_s, \tau_e]$, started at τ_s and happened, occurred or finished at τ_e.
A time annotated triple $f = \langle s, p, o \rangle [\tau_s, \tau_e]$ with $p \in P_f$ and $0 \leq \tau_s < \tau_e \leq \infty$ is called fact triple (or just fact), $\mathcal{F}$ is the set of all fact triples. We say f is valid at $[\tau_s, \tau_e]$, it started to be valid at τ_s and stopped to be valid at τ_e.
A basic triple $t = \langle s, p, o \rangle$ with $p \in P_s$ is called special triple, $\mathcal{S}$ is the set of all special triples, it is interpreted as an always valid fact triple $t = \langle s, p, o \rangle [0, \infty]$.
A time annotated triple $f = \langle s, p, o \rangle [\tau_s, \eta]$ is called a (current) state fact. A time annotated triple $f = \langle s, p, o \rangle [\tau_e, \tau_e]$ is a (current) state fact ending (event), i.e. if the system knows both $f = \langle s, p, o \rangle [\tau_s, \eta]$ and $f = \langle s, p, o \rangle [\tau_e, \tau_e]$ with $\tau_s < \tau_e \leq \eta$ then the current state fact ends, i.e. is replaced by $f = \langle s, p, o \rangle [\tau_s, \tau_e]$.
Note that facts are not true at time instants, but at the periods between time instants, so lightOn$[1, 2]$ and lightOff$[2, 3]$ is not contradictory as $[2, 2]$ is not an allowed time annotation for the validity of facts, only a possible time annotation for events, e.g. lightSwitchedOff$[2, 2]$ may have happened and be the cause for the fact change at time-point 2.

[1] A possible internal representation is milliseconds since system start, automatically converted from/into xs:dateTime [Malhotra et al., 2010] for outside communication.

[2] The current point of time is our main interest, we want to infer what is currently valid, neither what was true in the past nor what will (may) be true in the future.

[3] We use only URIs and literals and exclude blank nodes at moment due to inconsistencies in their interpretation [Mallea et al., 2011]

$\mathcal{T} = \mathcal{S} \cup \mathcal{E} \cup \mathcal{F}$ is the set of all possible triples. We use the functions $T(\mathcal{T}) \to T$, $\tau_s(\mathcal{T}) \to \tau$ and $\tau_e(\mathcal{T}) \to \tau$ to get the time annotation, the start time and the end time of triples. For all $s \in \mathcal{S}$ $T(s) = [0, \infty], \tau_s(s) = 0, \tau_e(s) = \infty$, i.e. their (implicit) annotation is since ever and for ever. A fact triple without given annotation is interpreted as $[0, \infty]$ annotated as well, if a not time annotated event enters the system it is annotated with $[now(), now()]$, if an event annotated with an time instant τ enters the system, its annotation is converted to the time stamp $[\tau, \tau]$.
The input of our system is a called triple stream, it is an infinite sequence of event triples $\mathcal{S} = e_1, e_2, ...$, where for any e_i, e_j in the stream $i \leq j \iff \tau_e(e_i) \leq \tau_e(e_j)$.

4.2 Triple Sets, Triple Pattern and Bindings

A triple pattern tp is a basic or annotated triple t that may contain variables V (symbols starting with ?) for any of the five annotated triple constituents, e.g. $\langle a, ?p, ?l \rangle$ and $\langle ?v, p, ?l \rangle [?t, \infty]$ are both triple patterns. We may reference to the variables contained in a triple pattern via the function $vars$.
A binding $\theta = \{?v_1/u_1, ?v_2/u_2, \ldots\}$ is a finite set of pairs of variables and terms/time instants, where every variable occurs at most once. A binding is applied to a triple pattern $tp\theta$ by replacing in the pattern all occurrences of variables in binding by their corresponding terms/time instants. A triple pattern tp matches a triple t with a binding θ, iff $tp\theta = t$.
We denote the set of all currently valid bindings with Ω.
Expressions are a way to bind new variables (allowed on left-side of = only) based on the execution of built-in functions on terms and already bound variables and to filter out bindings if all variables in the expression are already bound by the left context of the expression. Expressions are conjunctions of simple expressions, which are of the form: $t_0\{= \mid \neq \mid < \mid \leq \mid > \mid \geq\} f(t_1, \ldots, t_n)$ with $t_i \in V \cup L \cup T$.

4.3 Triple Graphs, Graph Pattern and Bindings

A temporal RDF graph is a finite subset of $\mathcal{T} = \mathcal{S} \cup \mathcal{E} \cup \mathcal{F}$.

Graph patterns are defined inductively:

- any triple pattern tp is a graph pattern; it is a fact graph pattern, if the predicate $p \in P_f$; it is an event graph pattern, if the predicate $p \in P_e$; it is either or, but not both - determined by its context - if the predicate is variable.
- If F, F_1, F_2 are fact graph pattern and E, E_1, E_2 are event graph pattern and B is a expression, then
  - $(F_1\ ,\ F_2)$,
  - $(F\ ,\ B)$,
  - $(F_1\ ;\ F_2)$ or equivalently $(F_1\ \mathit{union}\ F_2)$,
  - $(F_1\ \mathit{without}\ F_2)$,
  - $(F\ \mathit{without}\ E)$,
  - $(\mathit{since}\ E)$,
  - $(\mathit{until}\ E)$,
  - $(\mathit{replace}\ F\ \mathit{on}\ E)$,

– $(F\ on\ init)$

are fact graph pattern, and

– $(E_1\ ,\ E_2)$,
– $(E\ ,\ B)$,
– $(E_1\ ;\ E_2)$ or equivalently $(E_1\ union\ E_2)$,
– $(E_1\ without\ E_2)$,
– $(E\ without\ F)$,
– $(E_1\ seq\ E_2)$,
– $(E_1\ before\ E_2)$,
– $(E\ during\ F)$,
– $(E\ with\ F)$,
– $(onStart\ F)$,
– $(onEnd\ F)$

are event graph pattern. All other combinations are not admissible. A graph pattern is in TEF-SPARQL, if it is admissible.

Fact Pattern	$T(F)$	Conditions & Sideeffects	Returned Bindings
$< S, P_F, O > [\tau_s, \tau_e]$	$[\tau_s, \tau_e]$	$\tau_s < \tau_e$	$vars(< S, P_F, O > [\tau_s, \tau_e])$
F_1, F_2	$[max(\tau_s(F_1), \tau_s(F_2)),$ $min(\tau_e(F_1), \tau_e(F_2))$		$vars(F_1) \cup vars(F_2)$
F, B	$T(F)$		$vars(F) \cup vars(B)$
$F_1; F_2$ $F_1\ union\ F_2$	$T(F_1)\ or$ $T(F_2)$		$vars(F_1) \cap vars(F_2)$
$F_1\ without\ F_2$	$[max(\tau_s(F_1), \tau_e(F_2)),$ $\tau_e(F_1)]$	$\neg\exists F_2 \vee$ $\tau_e(F_1) > \tau_e(F_2)$	$vars(F_1)$
$F\ without\ E$	$[max(\tau_s(F), \tau_e(E)),$ $\tau_e(F)]$	$\neg\exists E \vee$ $\tau_e(F) > \tau_e(E)$	$vars(F)$
$since\ E$	$[\tau_e(E), \infty]$		$vars(E)$
$until\ E$	$[\tau_e(E), \tau_e(E)]$		$vars(E)$
$replace\ F\ on\ E$	$[\tau_e(E), \eta]$	if $\tau_e(F) = \eta$ then add $F[\tau_e(E), \tau_e(E)]$	$vars(F) \cup vars(E)$
$F\ on\ init$	$[0, \eta]$	$now() = 0$	$vars(F)$

Table 4.1: Fact Graph Pattern

Graph Pattern Matching Bindings for complex graph event/fact patterns are defined on graphs. A graph pattern gp matches a temporal graph G with a binding θ, iff $gp\theta = G'$ where $G' \subseteq G$. The time semantics are always defined in the context of a binding. As soon as the matching constraints of a graph pattern are fulfilled, the graph pattern produces a new binding. In the case of a fact graph pattern, its end time may not yet be determined. This is due to their interval-based time model. Consider, for example, a fact graph pattern whose end time is punctuated by an $untilE$ event graph pattern. As such, this binding may belong to a *partial match* for fact graph patterns. Event graph patterns produce complete matches only.

Event Pattern	$T(E)$	Conditions & Sideeffects	*Returned Bindings*
$< S, P_E, O > [\tau_s, \tau_e]$	$[\tau_s, \tau_e]$	$\tau_s \leq \tau_e$	$vars(< S, P_E, O > [\tau_s, \tau_s])$
E_1, E_2	$[min(\tau_s(E_1), \tau_s(E_2)),$ $max(\tau_e(E_1), \tau_e(E_2))]$	$\exists E_1 \wedge \exists E_2$	$vars(E_1) \cup vars(E_2)$
E, B	$T(E)$		$vars(E) \cup vars(B)$
$E_1; E_2$ E_1 *union* E_2	$T(E_1)$ *or* $T(E_2)$	$\exists E_1 \vee \exists E_2$	$vars(E_1) \cap vars(E_2)$
E_1 *without* E_2	$T(E_1)$	$\exists E_1 \wedge \neg\exists E_2 :$ $\tau_s(E_1) \leq$ $\tau_e(E_2) <$ $q\tau_e(E_1)$	$vars(E_1)$
E *without* F	$T(E)$	$\exists E \wedge \neg\exists F :$ $T(E) \cap T(F) \neq$ $\emptyset$	$vars(E_1)$
E_1 *seq* E_2	$[\tau_s(E_1), \tau_e(E_2)]$	$\exists E_1 \wedge \exists E_2 :$ $\tau_e(E_1) \leq$ $\tau_s(E_2)$	$vars(E_1) \cup vars(E_2)$
E_1 *before* E_2	$[min(\tau_s(E_1), \tau_s(E_2)),$ $\tau_e(E_2)]$	$\exists E_1 \wedge \exists E_2 :$ $\tau_e(E_1) \leq$ $\tau_e(E_2)$	$vars(E_1) \cup vars(E_2)$
E *during* F	$T(E)$	$\tau_s(F) \leq \tau_s(E)$ $\tau_e(E) \leq \tau_e(F)$	$vars(E) \cup vars(F)$
E *with* F	$T(E)$	$\tau_s(F) \leq \tau_s(E)$ $\leq \tau_e(F)$	$vars(E) \cup vars(F)$
onEnd F	$[\tau_e(F), \tau_e(F)]$		$vars(F)$
onStart F	$[\tau_s(F), \tau_s(F)]$		$vars(F)$

Table 4.2: Event Graph Pattern

Conjunction The graph pattern (F_1, F_2) matches G with a binding θ, iff

- there exist $\theta_1, \theta_2 \in \Omega$ for which F_1 and F_2 match some $g_1, g_2 \subseteq G$,
- the value for all $?v \in vars(F_1) \cap vars(F_2)$ is the same in θ_1 and θ_2, and
- $\Omega = \Omega \cup \{?v/u \in \theta_1 \quad | \quad ?v \in vars(F_1) \vee ?v \in vars(F_2)\}$.

The first condition ensures that both F_1 and F_2 match some triple of G. The second ensures that the join between F_1 and F_2 is valid. The third condition ensures that *all* variables make it into the final binding. Note that the join condition ensures that we can choose $?v/u$ from either θ_1 or θ_2, since $?v/u$ has the same value in both bindings.
The start time of a conjunction of events is the earliest starting time of both conjuncts whereas its ending time is the latest of these. In contrast, the start and end times for facts are defined as the latest starting and the earliest ending of the conjuncts, respectively.

Disjunction The graph pattern $(F_1; F_2)$ matches G with a binding θ, iff

- there exist $\theta_1, \theta_2 \in \Omega$ for which $F_1\theta_1$ or $F_2\theta_2$ match some $g_1, g_2 \subseteq G$,
- $\Omega = \Omega \cup \{?v/u \quad | \quad ?v/u \in \theta_1 \wedge ?v \in vars(F_1) \wedge ?v \in vars(F_2)\}$ (if $\theta_1 exists$), and
- $\Omega = \Omega \cup \{?v/u \quad | \quad ?v/u \in \theta_2 \wedge ?v \in vars(F_1) \wedge ?v \in vars(F_2)\}$ (if $\theta_2 exists$).

The first condition ensures that F_1 of F_2 indeed match. The second and third conditions ensure that if a binding exists it will be added to the set of bindings—for both F_1 and F_2. Note that the new bindings are restricted to the intersection of the variables that occur in F_1 and F_2. This restriction ensures first, that no unbound variables from one of the disjuncts is contained in the new binding. Second, under this regime outer joins are not admissible.
The time semantics of a disjunctive graph pattern is somewhat more complicated than for a conjunction. The start and end times are defined for the two possible graph patterns, separately. We illustrate this by the following example:

Example 4.3.1. *Assume the fact graph pattern that shall match whenever a person lives in a city or in a village:*

$$?xlivesInCity \quad UNION \quad ?xlivesInVillage$$

Note that this actually is an exclusive disjunction, since for each possible binding variables can only be bound to a single value. Never can both branches of such a disjunction match for any binding. In other words: observing a single triple from the stream of data we can never produce two bindings in the present case. The time fro which a match for a binding is valid solely depends on the validity of that very disjunct which matches for the binding. The produced binding contains only the common variables, in the present case $?x$.
Assume now the following fact graph pattern that can be used to find out the relations between that the entity $person_{123}$ has with the entities $City$ and $Village$:

$$person_{123}\, ?prop_1\, City \quad UNION \quad person_{123}\, ?prop_2\, Village$$

The disjunction does not contain common variables. Consequently, it matches, for example, for the binding $\theta = \{?prop1/livesIn, ?prop_2/likes\}$. The start time of the match depends on the earliest binding for $?prop1$ and $prop2$. Note that the binding produced b the match is actually empty as we take the intersection of the variables that occur in the disjunction.

Negation Negation for events and facts such as $E_1 without E_2$ means that E_2 either never occurs or it must have ended before E_1 ends. The time semantics is defined either as the whole interval of the positive fact/event E_1, in case the negative fact/event does not occur during the positive's lifetime. Or it is defined as the remaining time interval when the negated event/fact has already ended up until the positive event/fact will actually end.

Sequences For all temporal topological operations *seq*, $before$, *during*, and *with* we have that—as in the case of conjunction—the new binding contains all variables of the participating bindings. This reflects the immutable state of all bindings that conform a match not to change over time. The time-semantics are self-explanatory.

Punctuation For punctuation (*since*, *until*, *onEnd*, *onStart*) is it worth noting that the binding for an event E solely consists of the variables that are to be found in the fact F. Vice versa, the binding for a fact F solely consists of those variables that occur in E. We may hence "forget" about variable bindings via punctuation.

Filters Filters can be defined on both fact and event graph patterns. Their scope is limited to the binding of the very graph pattern that they are attached to. The matching times solely depend on that of the graph pattern and not of the matching time of the filter.

Binds Bind statements may bind values to variables. This can be useful, for example, when the result of the query variable can only be computed from the data. Consider, for example, the case for distinguishing the dividend between common stock and preferred stock. The latter usually has a higher dividend at the cost of no voting rights.

$$\begin{array}{l}(?x \quad type \quad CommonStock\,,\, ?x \quad hasValue \quad ?y) \quad BIND \quad ?y = 2{*}?x) \\ UNION \\ (?x \quad type \quad PreferredStock\,,\, ?x \quad hasValue \quad ?y) \quad BIND \quad ?y = 2{*}?x)\end{array}$$

Assume further that we are only interested in the value of $?y$. We then can use $BIND$ to declare the variable $?y$ and define its value as a functional outcome of the current binding, in our case the value for $?x$. The matching time of the $BIND$ solely depends upon that of the connectd graph pattern.

4.4 SELECT-Queries

SELECT-queries are defined similar to standard SPARQL.

```
SELECT ?v1 ... ?vN
ONCE PER ?v1 ... ?vM
WHERE {
  GP
}
```

In essence SELECT-queries are projection of the query variables $?v_n$ to the set of bindings Ω. The body of the query is defined by a graph pattern GP which can be a fact or an event graph pattern. We require each variable $?v_n$ to occur in GP at least once. The result of a SELECT-query is a stream of N-tuples. For every binding for which GP matches, the query produces one tuple. Each tuple consists of the ordered sequence of $?v_1, \ldots, ?v_N$ where each $?v_n$ has been substituted with a value in the binding, accordingly.
Similar standard SPARQL and SQL we can restrict results to be distinct. The ONCE PER statement restricts the result to one match for each distinct binding of the variables $?v_1, \ldots ?v_M$ where (i) each $?v_m$ is bound to a value and (ii) $\{?v_1, \ldots, ?v_M\} \subseteq \{?v_1, \ldots, ?v_N\}$, i.e. all variables of the ONCE PER statement have to occur in the projection. Note that the ONCE PER statement is optional.
For the sake of simplicity we assume that the stream of data we are matching against is unique. We consequently omit federated queries for this version of the language. Opposed to standard SPARQL, we cannot provide the source of the data specified by FROM statements. This will be added in a later version as it requires further investigation of the consequences. For the same reasons we omit a formal specification for prefixes.

4.5 Graph Pattern Construction

As in standard SPARQL, we support the construction of graph patterns using the CONSTRUCT operator.

```
CONSTRUCT {FACT|EVENT} GP_C
ONCE PER ?v1 ... ?vM
WHERE {
  GP
}
```

Similar to the case of SELECT-queries we do not construct N-tuples but graph patterns as defined by GPC. We restrict GPC to a list of triple patterns $TP_1, \ldots, TP_M$ where for all

predicates must solely drawn from either P_e or P_f. In other words: it is not admissible to produce events and facts at the same time. Optionally, i.e. for better readability, the type of tuples can be made explicte. The triple patterns may comprise IRIs and variables but not blank nodes. This imposes another restriction on the creation of new bindings: A binding θ that produces variable substitutions with values drawn from both P_e and P_f for a CONSTRUCT query Q, then that binding is not admissible for the query, if substituting the variables of the$Q$ the resulting triples had values from both P_e and P_f. As in the case of `SELECT`-queries for each binding for which GP matches we produce an instance of GP_C. The `ONCE PER` statement is defined in the same way. Prefixes and federation are also left for later specification.

4.6 Properties of the language

The most important property of a stream processing language is that of time-wise monotonicity Anicic et al. [2011]: "The querying formalism is intended to work on triple streams (i.e., triples continuously enter the system in the order of their associated time stamps) and query results are supposed to be output as soon as they are detected. This leads to the straightforward requirement that it should not be possible that query results once obtained get invalidated by later triple inputs."

Corollary. *The consequences of any fixed set of select and construct query in our language is monotone, if and only if the input stream is monotone.*

The dependence on a monotone stream is most obvious for the (E_1 *without* E_2) pattern. If finishing of E_1 arrives before a delayed E_2, the pattern is (wrongly) assumed to be satisfied at the end of E_1 and would need correction on the late arrival of E_2.
The monotonicity itself is a consequence of the following theorem, that stats that queries in our language can only express pattern, that involve the current point of time, i.e. it never allows to infer that something was true (or happened) in the past or will be true (will happen) in future. Consequently, nothing can depend on the future, nor contradict the past.

Theorem. *At any point in time (position of received events from the input stream), it only infers facts and events with* $E[\tau_s, \tau_e]$ *with* $\tau_e = now()$
$F[\tau_s, \tau_e]$ *with either* $\tau_s = now(),\ \tau_e = \eta$ *or* $\tau_s = \tau_e = now()$

The definition of the pattern in tables 4.1 and 4.2 ensure that.

Theorem. *For any event-stream and any finite set of queries in our language without generation of new terms (URI and type elements) nor blank-nodes in the head of construct statement it is decidable wether a triple is in the answer-set and at any point in time and the number of triple is in the answer-set is finite.*

Up to any point in time, there is only a finite number of time points, and as the language cannot infer statements about the future the maximum number of annotated triples is $|I|^2 * |I \cup L| * now()^2/2$. With a finite number of possible triples the question, if a triple is in the answer-set is decidable.

Theorem. *Deciding wether a purely conjunctive graph pattern without time annotations matches a graph is NP-hard.*

This is equivalent (as triple and nary-relations are equivalent) to the theta-subsumption problem in Datalog clauses, e.g. see Section 3 in Kietz and Lübbe [1994] for a proof of the NP-hardness of theta-subsumption, that can be directly used for conjunctive graph pattern.

Theorem. *Deciding wether a purely conjunctive graph pattern with time annotations pattern as defined in section 4.3 matches a graph is within NP.*

If we can guess the θ of the match correctly, the verification is a simple subset test and can be performed in polynomial time.
For graph pattern containing complex *without* and *union* subexpression this simple verification cannot be used, Their complexity is likely higher.

Chapter 5

Use-cases for the language

5.1 Using Events to cache Never/Slow-Changing Facts in States/Events

Consider you want to recommend TV-Movies to people. For that you need to know things like what kind of movie it is, how good it is rate why which groups of people, which actors play in the movie, and so on. All these information are available in the internet, but querying the internet for them takes a while. Clearly, the events ?Channel start ?Movie and ?Channel finishes ?Movie are coming over the stream. But you want to recommend movies close to their start and not when they are running already for a while. So querying the internet on demand takes much to long and keeping a local copy of the (potentially relevant parts of the) internet to get faster answers is not realistic as well.
Fortunately the schedule of movies on TV is known before so it is not to difficult to have an event ?Channel willShowNext ?Movie sufficiently ahead of time to query the internet and keep the result of this query in the engine as state or event, till the movie is finished. This can be done with a query like the following:

```
CONSTRUCT <Things you want to cache about the movie>
     WHERE ((SINCE ?Channel willShowNext ?Movie)
            UNION
            (TILL ?Channel finishes ?Movie)
           )
           DURING
           <Things you need to query from the internet>
```

For most parts the schedule is also known weeks ahead and can be accessed before, i.e.. facts for the pattern `?Show isScheduledToStartAt ?DateTime` can be cached with a similar query as above from the internet at least `?DateTime - "PT1H" = now()`. Given that, we can replace the dependence on the external event `(SINCE ?Channel willShowNext ?Movie)` by a time driven pattern that let us catch things 1 hour before scheduled start `(?Show isScheduledToStartAt ?DateTime, now() > ?DateTime - "PT1H")`.

5.2 Using Facts to compute arbitrary aggregations

Most Stream Engines provide aggregation functions (similar to SQL) over time windows. If the needed function is a provided function and the specification of a time or size window is adequate for the problem, they are easy to use and efficient computable.
However, when the sliding window is not a adequate way to characterize the relevant data, then their usage become clumsy, inefficient or even impossible.

In EP-SPARQL (as in our language) no specific aggregate functions functions exist. But nevertheless aggregations can be computed. [Anicic et al., 2011] present the following example query to illustrate a moving average computation in EP-SPARQL.

```
CONSTRUCT _:aaa :hasCount ?count .
          _:aaa :hasSum ?sum .
   { SELECT ?count AS ?prevcount + 1
            ?sum AS ?prevsum + ?price
      WHERE {{ ?point :hasCount ?prevcount .
               ?point :hasSum   ?prevsum .
             } SEQ { :ACME :hasStockPrice ?price . }
            } EQUALSOPTIONAL
            {{ ?point :hasCount ?prevcount .
               ?point :hasSum ?prevsum .
             } SEQ { :ACME :hasStockPrice ?inbetween .
             } SEQ { :ACME :hasStockPrice ?price . }
            }
     FILTER ( !BOUND(?inbetween) &&
              getDURATION() < "P10D"^^xsd:duration )}

SELECT ?sum / ?count AS ?average
 WHERE {{ :ACME :hasStockPrice ?price . }
       SEQ { ?point :hasCount ?point :hasSum ?prevsum . }
       } EQUALSOPTIONAL
       {{ :ACME :hasStockPrice ?price . }
        SEQ { :ACME :hasStockPrice ?inbetween . }
        SEQ { ?point :hasCount ?prevcount .
              ?point :hasSum ?prevsum . }
            }
 FILTER ( !BOUND(?inbetween) &&
          getDURATION() > "P10D"^^xsd:duration )
```

We think the EP-SPARQL formulation of this query has three disadvantages compared to an formulation in our language. The query uses generation of new URI via blank nodes in the head of the `CONSTRUCT`, this enables an infinite number of possible triples and as the query simultaneously uses recursion[1] (`_:aaa :hasSum ?sum` is contructed out of `?point :hasSum ?prevsum` the an EP-SPARQL formalism enabling this query is most likely undecidable. The complicated `EQUALSOPTIONAL` part together with the `!BOUND(?inbetween)` filter is the clumsy way how EP-SPARQL supports negation. The in our view most clumsy thing in this query is however that EP-SPARQL only has events and misses time-depended facts. The `:hasCount` and `:hasSum` are de facto facts of limited validity, i.e. any is are true between two `:ACME :hasStockPrice ?price` events and they are not events that occur in between as modeled in this EP-SPARQL query. If n `:ACME :hasStockPrice ?price` events occur inside the window, then this query has to keep $\sum_{i=1}^{n-1} 3i = 1.5 * n * (n-1)$ triples inside partial matches in memory to work correctly in all cases.
In our language that combines occurring events with facts limited for a period of time, the above behavior can be reached easier and with less memory.

```
CONSTRUCT ?Ticker sumMember ?P,
          ?Ticker sum ?NSum,
          ?Ticker count ?NCount,
```

[1] It is probaly even wrong as the recursion base case `?point :hasCount 0 .  ?point :hasSum 0` at the start of each window is missing.

```
          ?Ticker tenDayAverage ?NSum/?NCount
    WHERE REPLACE(?Ticker sum ?Sum,
                  ?Ticker count ?Count.
                  ?Ticker tenDayAverage ?_)
               ON (?Ticker :hasStockPrice ?price[?_,?TN],
                   WITHOUT (Ticker sumMember ?OP[?TO,?_],
                            duration(?TO, ?TN) > 10d),
          ?NSum = Sum + ?price, ?NCount = Count +1)

CONSTRUCT ?Ticker sum ?NSum,
          ?Ticker count ?NCount
    WHERE (REPLACE(Ticker sumMember ?OP[?TO,?_],
                  ?Ticker sum ?Sum,
                  ?Ticker count ?Count)
               ON (?Ticker :hasStockPrice ?_[?_,?TN,
                   duration(?TO, ?TN) > 10d),
          ?NSum = ?Sum - ?OP, ?NCount = ?Count -1
```

The main trick in our query is to remember the currently valid state, i.e. sum and count and all the `?prices` that are in the aggregation, as facts. This allows us to subtract them from the current sum if they become to old to be in the window any longer. If the maximum of prices inside the time-window is n, the minimum[2] memory needed is $n + 4$, i.e. the n elements in the window, the new event, that let some old ones drop out of the window and a triple for each of sum, count and average. When a new price event occurs the new price is added the update occurs. The first `CONSTRUCT` adds the new price as member element, to the sum and to the count, but only if no (now) outdated sumMember exist (or has been subtracted by the second query). In this case the new average is computed as well. If there are members which are now to old to be kept in the window, they are subtractd by the second `CONSTRUCT` This query terminates all (one after the other, till now is found) members which are to old and reduces sum and count.
The first query needs to keep partial matches for: 1 sum fact, 1 count fact, 1 tenDayAverage fact and 1 hasStockPrice event the second query needs to keep partial matches for: N sumMember facts, 1 sum fact, 1 count fact and 1 hasStockPrice event. which gives a total of $N + 7$ triples to be stored as partial matches, i.e. is optimal up to a constant summand (not even factor) and as a second advantage over EP-SPARQL, it does not need to generate new URI to do so, i.e. by avoiding this source of infinity, we reduce the danger of undecidability.[3]
This illustrates how arbitrary time-window based aggregations can be handled in our language efficiently. But it is able to do more, namely aggregate in event-delimited windows. Let's look on such an even-delimited aggregation. Let's assume we get the following events about TV-shows and TV-viewer. TV-shows are delimited by 2 events on the stream `?Channel starts ?Show` and `?Channel finishes ?Show`, and when a TV-viewer switches on or off or between different channels we get `?Person joins ?Channel` and `?Person leaves ?Channel` events. Let us compute to example aggregations, namely "the maximum and the average number of viewers of a TV-show". This is also a slightly more complicated average, just taking the mean value of all the different currentNoOfViewers numbers of a channel during a show does not capture an essential dimension, namely

[2]The only way to reduce this memory requirement is to bins of several values added and removed together to the moving average, i.e. to trade precision for memory.

[3]We still use numbers and arithmetic to build new numbers, but this well know source of infinity can be much better controlled than a possibly infinite number of URI's.

how long how many were watching the show. So the currentNoOfViewers weighted by is validity duration captures the intuition much better.

```
CONSTRUCT FACT ?Channel currentNoOfViewers ?N WHERE
   (REPLACE ?Channel currentNoOfViewers ?N1
         ON ?_ joins ?Channel), ?N = ?N1+1
   UNION
   (REPLACE ?Channel currentNoOfViewers ?N2
         ON ?_ leaves ?Channel), ?N = ?N2-1
   UNION
   ((?Channel rdf:type TV-Channel, ?N = 0) ON INIT)

CONSTRUCT FACT ?Channel shows ?Show WHERE
   (SINCE ?Channel starts ?Show)
   UNION
   (UNTIL ?Channel finishes ?Show)

CONSTRUCT EVENT ?Show maximumViewer ?N WHERE
   (((?Channel starts ?Show
      BEFORE ?Channel finishes ?Show
     ) WITH (?Channel currentNoOfViewers ?N)
    ) WITHOUT
      ((ONSTART ?Channel currentNoOfViewers ?HN)
       DURING ?Channel shows ?Show, ?HN > ?N)
   )

CONSTRUCT EVENT ?Show averageNoOfViewers ?N WHERE
   ((?Channel starts ?Show),
    ((?Channel finishes ?Show)
     WITH (?Show weightedSumOfViewers ?S)))[?ST,?ET],
    ?N = ?S/duration(?ST,?ET)

CONSTRUCT FACT ?Show weightedSumOfViewers ?S WHERE
 (REPLACE ?Channel weightedSumOfViewers ?OS
       ON (ONEND ?Channel currentNoOfViewers ?N[?ST,?ET]),
          ?S = ?OS + ?N*duration(?ST,?ET))
 UNION
 ((SINCE ?Channel starts ?Show, ?S = 0)
```

5.3 Using precise queries to generate only the interesting matches

Declarative languages like EP-SPQRL have an inherent problem with multiple matches and a resulting explosion of match candidates to be kept and answers to be generated. If you take the example query from [Anicic et al., 2011] and assume a stream of data, containing the daily end-price of stocks as shown in Fig. 5.1, the problem becomes obvious. How many partial matches have to be kept by the engine and how many complex events have to be signaled at which time with which time-interval to the user.

1. once at day21 (but with which of the 10 possible intervals?) and never again or
2. once at day21 (but with which of the 10 possible intervals?) , once at day22, ..., once at day30

```
IBM hasStockPrice 1.00 at day1
...
IBM hasStockPrice 0.98 at day10
IBM hasStockPrice 0.50 at day11
...
IBM hasStockPrice 0.54 at day20
IBM hasStockPrice 1.20 at day21
...
IBM hasStockPrice 1.24 at day30

CONSTRUCT ?company OWL:ISA VolatileShare WHERE
     { ?company hasStockPrice ?price1 }
 SEQ { ?company hasStockPrice ?price2 }
 SEQ { ?company hasStockPrice ?price3 }
FILTER ( ?price2 < ?price1 * 0.7 &&
         ?price3 > ?price1 * 1.05 &&
         getDURATION() < "P30D"^^xsd:duration)
```

Figure 5.1: Stock market example

3. 10 times at day21 (with intervals [day1,day21], [day2,day21], ..., [day10,day21]), 10 times at day22, ..., 10 times at day30

4. 100 times at day21, 100 times at day22, ..., 100 times at day30

If the result of this complex event definition is used as input by another query or event only 3) and 4) ensure the completeness of any possible followup-query. However, if the result of this query is signaled to a human only 1) or 2) seems to be acceptable as humans don't like to get overwhelmed with more or less redundant events.
A usual way to reduce the answerset from case 4) to case 3) is the usage of `distinct` keyword within the `CONSTRUCT/SELECT`. With EP-SPQRL it is not possible to formulate a query with the behavior of case 1) or case 2). Our language allows a more precise formulation of the query, i.e. to specify that this should be generated only once per company (case 1) or once per companay and day (case 2)

```
ONCE PER ?company {, fn:day-from-dateTime(now())}
```

Chapter 6

Conclusion

In this report we presented an extension of SPARQL for processing dynamic and time-annotated data, namely TEF-SPARQL. To the best of our knowledge it is first language to respect the semantic difference between temporal facts and events and to provide useful and sound operators for their interaction. Special focus is laid on the combination of time relations and joins. To avoid potential complexity issues, only a limited form of negation can be expressed. The combination of facts and events allows for expressive aggregations. TEF-SPARQL has been structured for operation on data driven episodes. The implementation of the actual matching process will hence base on Event-Processing approaches. A realization of a distributed event processor has already started. In a subsequent step we will build a compiler that will translate a given query into the execution framework.
The design of the language follows the path of combining expressive power with efficient distributable implementation. TEF-SPARQL in particular respects an efficient potential implementation of aggregates. The actual evaluation will take place in phases two and three. This evaluation will be based on three sample use-cases. Adoptions to the language—if necessary—will follow a thorough analysis of the results.

Acknowledgements

The authors gratefully acknowledge the Office of Naval Research for the support of this work under the *The Naval International Cooperative Opportunities in Science and Technology Program (NICOP)*, grant number N62909-11-1-7065.

Bibliography

R. Angles and C. Gutierrez. The Expressive Power of SPARQL. In A. Shet, S. Staab, M. Dean, M. Paolucci, D. Maynard, T. Finin, and K. Thirunarayan, editors, *The Semantic Web - ISWC 2008: 7th International Semantic Web Conference, ISWC 2008, Karlsruhe, Germany, October 26-30, 2008, Proceedings*, volume 5318 of *LNCS*, pages 114–129. Springer, 2008.

D. Anicic, P. Fodor, and S. Rudolph. A rule-based language for complex event processing and reasoning. In *Web Reasoning and Rule Systems*, volume 6333 of *Lecture Notes in Computer Science*, pages 42–57. Springer, 2010. URL `http://www.springerlink.com/index/5G22262881813263.pdf`.

D. Anicic, P. Fodor, S. Rudolph, and N. Stojanovic. EP-SPARQL: a unified language for event processing and stream reasoning. In *Proceedings of the 20th International Conference on World Wide Web*, pages 635–644. ACM, 2011. URL `http://portal.acm.org/citation.cfm?id=1963495`.

A. Arasu, B. Babcock, S. Babu, J. Cieslewicz, M. Datar, K. Ito, R. Motwani, U. Srivastava, and J. Widom. STREAM : The Stanford Data Stream Management System. Technical report, 2004.

A. Arasu, S. Babu, and J. Widom. The CQL continuous query language: semantic foundations and query execution. *The VLDB Journal*, 15(2):121–142, July 2005. ISSN 1066-8888. doi: 10.1007/s00778-004-0147-z. URL `http://www.springerlink.com/index/10.1007/s00778-004-0147-z`.

D. Barbieri, D. Braga, S. Ceri, and M. Grossniklaus. An execution environment for C-SPARQL queries. In *Proceedings of the 13th International Conference on Extending Database Technology*, pages 441–452. ACM, 2010. URL `http://portal.acm.org/citation.cfm?id=1739041.1739095`.

D. D. F. Barbieri, D. Braga, S. Ceri, E. Della Valle, and M. Grossniklaus. C-SPARQL: SPARQL for continuous querying. In *Proceedings of the 18th International Conference on the World Wide Web*, volume 427 of *WWW '09*, pages 1061–1062. ACM, 2009. ISBN 9781605584874. doi: 10.1145/1526709.1526856. URL `http://www2009.org/proceedings/pdf/p1061.pdfhttp://portal.acm.org/citation.cfm?id=1526856`.

R. S. Barga, J. Goldstein, M. Ali, and M. Hong. Consistent Streaming Through Time : A Vision for Event Stream Processing. In *CIDR 2007, Third Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 7-10, 2007*. CIDR, 2007.

C. Bizer, T. Heath, and T. Berners-Lee. Linked Data - The Story So Far. *International Journal on Semantic Web and Information Systems*, 5(3):1–22, 2009. URL `http://www.citeulike.org/user/omunoz/article/5008761`.

L. Brenna, A. Demers, J. Gehrke, M. Hong, J. Ossher, B. Panda, M. Riedewald, M. Thatte, and W. White. Cayuga: a high-performance event processing engine. In *SIGMOD Conference*, pages 1100–1102, BEIJING, 2007. ACM. ISBN 9781595936868. doi: 10.1145/1247480.1247620. URL `http://portal.acm.org/citation.cfm?id=1247480.1247620`.

S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, V. Raman, F. Reiss, and M. Shah. TelegraphCQ: Continuous Dataflow Processing for an Uncertain World. *CIDR*, 20(March):668, 2003. doi: 10.1145/872757.872857. URL `http://db.cs.berkeley.edu/papers/cidr03-tcq.pdf`.

A. Demers, J. Gehrke, M. Hong, and M. Riedewald. Towards Expressive Publish / Subcribe Systems. In Y. E. Ioannidis, M. H. Scholl, J. W. Schmidt, F. Matthes, M. Hatzopoulos, K. Böhm, A. Kemper, T. Grust, and C. Böhm, editors, *Advances in Database Technology - EDBT 2006, 10th International Conference on Extending Database Technology, Munich, Germany, March 26-31, 2006, Proceedings*, volume 3896 of *Lecture Notes in Computer Science (LNCS)*, pages 1–18. Springer, 2006. ISBN 3-540-32960-9.

A. Demers, J. Gehrke, M. Hong, B. Panda, M. Riedewald, V. Sharma, and W. White. Cayuga: A General Purpose Event Monitoring System. In *CIDR 2007, Third Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 7-10, 2007*, pages 412–422. CIDR, 2007.

Y. Diao, N. Immerman, and D. Gyllstrom. SASE+: An Agile Language for Kleene Closure over Event Streams. Technical report, 2007.

S. Harris and A. Seaborne. SPARQL 1.1 Query Language. Technical report, The World Wide Web Consortium (W3C), 2012. URL `http://www.w3.org/TR/sparql11-query/`.

J. Hoeksema and S. Kotoulas. High-performance Distributed Stream Reasoning using S4. In *FIRST INTERNATIONAL WORKSHOP ON ORDERING AND REASONING*, 2011.

J.-U. Kietz and M. Lübbe. An Efficient Subsumption Algorithm for Inductive Logic Programming. In *ICML*, pages 130–138, 1994. URL `http://dblp.uni-trier.de/db/conf/icml/icml1994.html\#KietzL94`.

G. Klyne and J. J. Carroll. Resource Description Framework (RDF): Concepts and Abstract Syntax, 2004. URL `http://www.w3.org/TR/rdf-concepts/`.

J. F. Lajos, G. Toth, R. Racz, J. Panczel, T. Gergely, and A. Beszedes. Survey on Complex Event Processing and Predictive Analytics. Technical report, Citeseer, 2010. URL `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.173.2029\&rep=rep1\&type=pdf`.

D. Maier, M. Grossniklaus, S. Moorthy, and K. Tufte. Capturing Episodes : May the Frame Be with You. In *DEBS 2012*, 2012. ISBN 9781450313155.

A. Malhotra, J. Melton, N. Walsh, and M. Kay. XQuery 1 . 0 and XPath 2 . 0 Functions and Operators (Second Edition), 2010. URL `http://www.w3.org/TR/xpath-functions/`.

A. Mallea, M. Arenas, A. Hogan, and A. Polleres. On Blank Nodes. In *ISWC*, volume 1380, pages 421–437, 2011.

Y. Mei and S. Madden. ZStream : A Cost-based Query Processor for Adaptively Detecting Composite Events Categories and Subject Descriptors. *Proceedings of the 35th SIGMOD international conference on Management of data*, pages:193–206, 2009. doi: 10.1145/1559845.1559867. URL http://portal.acm.org/citation.cfm?id=1559867.

T. Owens. Survey of event processing. *Distribution*, (December), 2007. URL http://oai.dtic.mil/oai/oai?verb=getRecord\&metadataPrefix=html\&identifier=ADA475386.

M. Perry and A. P. Sheth. SPARQL-ST : Extending SPARQL to Support Spatiotemporal Queries. *Design*, 2009.

M. Stonebraker and U. Cetintemel. "One Size Fits All": An Idea Whose Time Has Come and Gone. *21st International Conference on Data Engineering ICDE05*, 0(Icde):2–11, 2005. ISSN 10844627. doi: 10.1109/ICDE.2005.1. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1410100.

E. Wu, Y. Diao, and S. Rizvi. High-performance complex event processing over streams. *Proceedings of the 2006 ACM SIGMOD international conference on Management of data - SIGMOD '06*, page 407, 2006. doi: 10.1145/1142473.1142520. URL http://portal.acm.org/citation.cfm?doid=1142473.1142520.