SPECTRA: Continuous Query Processing for RDF Graph Streams Over Sliding Windows

Syed Gillani
Univ Lyon
UJM Saint-Étienne
CNRS
Laboratoire Hubert Curien
Saint-Étienne, France
syed.gillani@univ-st-etienne.fr

Gauthier Picard
Univ Lyon
MINES Saint-Étienne
CNRS
Laboratoire Hubert Curien
Saint-Étienne, France
picard@emse.fr

Frédérique Laforest
Univ Lyon
UJM Saint-Étienne
CNRS
Laboratoire Hubert Curien
Saint-Étienne, France
frederique.laforest@univ-st-etienne.fr

ABSTRACT

This paper proposes a new approach for the the incremental evaluation of RDF graph streams over sliding windows. Our system, called "SPECTRA", combines a novel form of RDF graph summarisation, a new incremental evaluation method and adaptive indexing techniques. We materialise the summarised graph from each event using vertically partitioned views to facilitate the fast hash-joins for all types of queries. Our incremental and adaptive indexing is a byproduct of query processing, and thus provides considerable advantages over offline and online indexing. Furthermore, contrary to the existing approaches, we employ incremental evaluation of triples within a window. This results in considerable reduction in response time, while cutting the unnecessary cost imposed by recomputation models for each triple insertion and eviction within a defined window. We show that our resulting system is able to cope with complex queries and datasets with clear benefits. Our experimental results on both synthetic and real-world datasets show up to an order of magnitude of performance improvements as compared to state-of-the-art systems.

CCS Concepts

•**Information systems** → **Graph-based database models; Data streams; Query optimization;** *Main memory engines;*

Keywords

RDF Graphs; Incremental Stream Processing; sliding windows

1. INTRODUCTION

An increasing number of scientific applications are storing and disseminating their data sets on the Web as Linked Open Data[1] using Resource Description Framework (RDF) [34]; thus facilitating data reusability and interoperability. For instance RDF Data Cube[2] is a W3C recommended vocabulary to publish multi-dimensional statistical data, and Linked Open science[3] is a platform to publish scientific data. RDF data are modelled in terms of a set of triples $\langle subject, predicate, object\rangle$ (or $\langle s, p, o\rangle$ for short), and intrinsically form a set of labelled and directed multigraphs; and SPARQL[4] is the most common query language for RDF data. With the increasing number of both commercial and non-commercial organisations which actively publish RDF data, the amount and diversity of openly available RDF data is rapidly growing.

As the volume of RDF data is continuously soaring, managing, indexing and querying very large collections of RDF data has become challenging. One approach to handle such large RDF data graphs is to process them using the *data stream* model [6, 8, 23], where streams of RDF data are processed within a predefined window. In such a model, recent elements of a stream are more important than those that arrived a long time ago: older objects are not of main concern and thus dropped. This preference for recent elements is commonly expressed using a sliding window [6], which identifies a portion of the stream that arrived between "now" and some recent time in the past. Algorithms for the so-called "streaming model" [6] must process the incoming graphs as they arrive, while bounding a set of them within a time/count-based window. Such constraints also capture various properties that arise while processing data for dynamic domains, such as sensor networks, social networks, geospatial systems etc., and ensure I/O efficiency when data do not fit into the main memory. RDF data in streaming environments, called *RDF graph streams*, are dynamic and updated continuously.

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

This paper introduces SPECTRA, an in-memory framework that tackles the challenge of continuously processing RDF graph streams in an incremental manner. As a framework, SPECTRA combines RDF graph summarisation and an efficient data structure – called *query conductor* – with an incremental and adaptive indexing technique to match a set of RDF graphs within a sliding window. To

[1] http://linkeddata.org

SSDBM '16 Budapest, Hungary

DOI:

[2] https://www.w3.org/TR/vocab-data-cube/\#intro-rdf

[3] http://linkedscience.org/data/

[4] http://www.w3.org/TR/rdf-sparql-query/

avoid storing and processing all the graph objects from the streams, we exploit the structure of the query to prune irrelevant information. That is, the registered query is used to prune all the triples that do not match the subjects, predicates and objects of the patterns defined in the query. The pruned set of triples, called *summary graph*, is used to implement multi-way joins between the set of defined patterns in the query. This results in pruning all the invalid triples without incurring storage and query processing costs. Summarised RDF graph events are materialised into a set of vertically partitioned [1] *views*, where each view – a two-column table (s, o) – stores all the information for each unique predicate in the summarised RDF graph. Here we use our incremental indexing technique inspired from the database cracking [19, 20] to index the joined (s, o) pairs within the set of views. It is a fully dynamic approach as it assumes no workload knowledge and requires no idle time for its creation; indices are built continuously, partially and incrementally as part of the query processing. A set of views represents the universe of the triples to be matched, and hash-join operations between views are used to join the triples based on subject/object column. The joined triples are incrementally indexed using a sibling relationship between them, thus enabling SPECTRA to support complex queries. A *timelist* is also used to associate the indexed triples with their respective timestamps; this permits the system to detect the older matches, as the window slides. Our experimental analysis confirms the superiority of our method and shows that SPECTRA outperforms state-of-the-art solutions upto an order of magnitude.

The rest of the paper is structured as follows. Section 2 presents the related work and limitations of the existing techniques, Section 3 describes the formal concepts and problem definition, Section 4 provides an overview of the framework, Section 5 presents the graph summarisation technique, Section 6 and 7 present the continuous and incremental query processing techniques respectively, Section 8 details the experimental evaluation and Section 9 concludes the paper.

2. RELATED WORK

This section first reviews the related work on querying static RDF graphs and RDF stream processing (RSP) systems and then outline their limitations.

Naively, one could approach the problem of processing RDF graph streams by leveraging existing RDF solutions [4, 27, 33, 35] for static data. That is, by (1) storing the entire stream, and (2) running queries for every incoming RDF graph. Clearly, this naive approach has severe limitations and is not in line with the streaming model [6]. First, storing streams obviously contradicts the idea of stream processing. Second, existing techniques utilise *offline indexing* [12], i.e., assuming enough workload knowledge and idle time to build the physical design before queries arrive to the system. This results in extensive indexing and expensive preprocessing of RDF graph data that may add considerable delay for each graph arriving in the stream. The third issue is the expensive recomputation of the query results under triple/graph arrival and eviction within a defined window.

Another line of solutions that may be helpful in our tasks is called *RDF stream processing* (RSP) systems [8, 23, 21, 11]. These solutions comply to the streaming model, albeit in an different manner. They are based on Data Stream Management Systems (DSMS) [7] for un/semi-structured data streams, where each element within a stream consists of a triple, and the system has to construct the matched graphs from a set of triple streams. These solutions, as discussed later, entail expensive constraints for processing RDF graph streams, and employ *online indexing* [30] techniques. Although, online indexing makes a step towards the more dynamic environments by allowing for continuous monitoring and periodically evaluating the index design, its performance degrades with variable workloads and increase in the size of triples within a defined window. Furthermore, most of the existing RSP systems suffer from the same problem of recomputation of the results from scratch; with the arrival or eviction of a triple, all the triples within a window are recomputed.

2.1 Static and Stream-based RDF Querying

Static RDF Query Processing. Most of the existing solutions for querying static RDF graphs [4, 27, 33, 35] – that differ in underlying storage structure and type of indexing – match a query with the RDF dataset in two steps. The first step retrieves a candidate set of graphs that contains the indexed features of the query. The second step uses subgraph isomorphism (subsequently homomorphism) to validate each candidate graph against the defined query. The majority of existing approaches – both in-memory and disk-based – follows a relational approach for storing and indexing RDF data. Approaches such as SW-store [1] vertically partition RDF triples into a set of tables and use an index structure on top of it to locate the required tables. Hexastore [32] and RDF-3X [27] employ index-based solutions by storing triples directly in B+-trees over multiple, redundant $\langle s, p, o\rangle$ permutations, while Jena [33] and Sesame [10] exploit multiple-property tables. Coupled with sophisticated statistics and query-optimisation techniques, these index-based approaches are very competitive for static settings.

RDF Stream Processing. The standardisation of RDF-based stream processing is still an ongoing debate and the W3C RSP community group[5] is an important initiative in this context. Many RSP solutions [8, 11, 21, 23] have evolved during the last few years and most of them are based on the triple stream model, i.e, a triple $\langle s, p, o\rangle\,[\tau]$ associated with a timestamp (τ) is used as an input to the system. These systems utilise various custom query languages – extended from SPARQL with temporal constructs – to match the defined pattern on a set of triples from a set of streams. The triples set within a window is later indexed to optimise the pattern matching. For instance, CQELS [23] uses a B+-tree to index the triples within a window, while C-SPARQL [8] uses the underlying Jena architecture to store and index triples within property tables. CQELS, which is considered as the most optimised system of the lot, employs an online indexing technique using Eddy operators [5] and query optimisations on top of the DSMS. SparkWave [21] uses the RETE network to determine the set of triggers to fire when a new triple arrives. Thus, it materialises intermediate results to reduce the amount of work required for each update. However, it works on a fixed schema for the stream, and it cannot handle complex queries that contain cyclic or tree-like patterns.

General-labelled Graph Streams. Some works related to the graph streams exist in the context of general labelled-graphs. Choudhury *et al*. [14] provides a subgraph selectivity approach to determine subgraph search strategies. It uses a subgraph-tree structure to decompose the query graph in smaller subgraphs, which is responsible for storing partial results. Finally the partial results are incrementally joined by searching through the tree, where incremental updates can be processed quickly. However, keeping and querying thousands of edges within tree nodes for a large window requires huge amount of space and computational resources. Moreover, this approach only supports simple path-based queries and is only optimised for homogeneous graphs by utilising an edge stream model. The work by Fan *et al*. [15] presents algorithms for graph pattern matching over evolving graphs by employing repeated

[5] `https://www.w3.org/community/rsp/`

search strategy to compute matches until a fixed point is reached – for each graph update and removal. They do not cater the streaming nature of the input data neither sliding windows. Chen *et al.* [13] propose a feature structure called the node-neighbour tree to search multiple graph streams using a vector space approach. They relax the exact match requirement and their system requires significant preprocessing on the graph streams.

While the above mentioned techniques implement efficient online query processing for general-labelled graphs, the major difference is the problem statement: the RDF data model is not directly isomorphic with the notion of directed labelled-graphs; in particular, edge labels can themselves be vertices. Therefore, in terms of performance, these solutions are not directly comparable with specialised RDF storage and querying systems (see [16] for details); they do not scale well to very large RDF graphs and complex queries. In this paper, our focus is to provide a specialised system for RDF graph streams and adapted for the sliding window model.

2.2 Limitations of Existing Solutions

Analysing related work shows that certain issues recur, here we summarise recurring limitations in detail.

Offline/Online Indexing. Offline indexing [12] techniques, as used by static RDF solutions, create indices a priori assuming accurate workload knowledge and data statistics; and plenty of priori slack time to invest in physical design. But, in the context of dynamic streaming environments, such knowledge and complete dataset cannot be known a priori. Moreover, traditional indices on static RDF triples cover all triples equally, even if some triples are needed often and some never. For instance, RDF3x [27] builds several clustered B+trees for all the permutations of $\langle s, p, o\rangle$ and has a time complexity $\mathcal{O}(n)$ for index creation/update with n the number of triples. Online indexing [8, 23] tackles some of the above mentioned issues and is employed by the RSP systems. The general idea is that the basic concepts of offline indices are transferred online. That is, while processing queries the system monitors the workloads and performance, it questions the need of different indices and once certain thresholds are passed, it triggers the creation of new indices and drops old ones. Such techniques perform better than offline indices in dynamic settings. However, in case of variable workloads, the creation of new indices from scratch can considerably outweigh the cost of query processing. This requires incremental indexing technique, where index creation and re-organisation take place automatically and incrementally.

Match Recomputation. The recomputation/re-evaluation of matches, once the data are updated within a window, can result in unnecessary utilisation of computation resources. Therefore, the challenge is to develop an incremental query processing, where the new query matches are computed by utilising previous query results, and the window is refreshed by only considering the effective area of the older matches. Most of the existing techniques for RSP are based on a recomputation model, i.e., with the insertion or eviction of triples in a window, query results are recomputed. As a consequence, these systems suffer from significant performance loss, as shown in our experimental study (Section 8).

Limited Scope. Existing RSP systems are evolved from the DSMS for un/semi-structured data; hence, the use of the triple-based streaming model was an obvious choice. However, as RDF data are graphs, it is not desirable to place any limitation on the event model of RDF graph streams. The consumption of RDF graphs as triples would tear-up the joined data graph and would result in extra computation overheads for each triple update. For instance, Eddy operators [5] employed by CQELS, which are inherited from the relational DSMS, result in expensive computation and continuously devote resources to explore all the plans (for each input triple) and require fully pipelined execution for RDF streams. Thus, caching the statistical measure of triples and choosing a right order for every triple update causes a huge overhead (as shown in our experimental analysis).

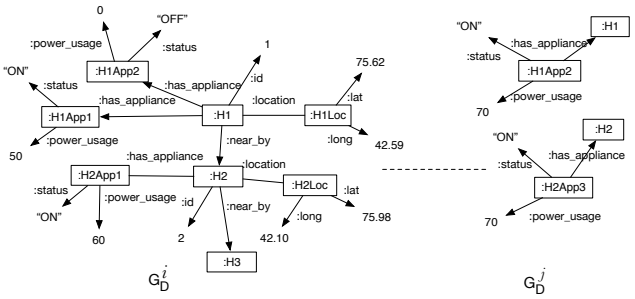

Figure 1 Two RDF Graph Events G_D^i and G_D^j

3. FORMAL CONCEPTS AND PROBLEM DEFINITION

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

An RDF graph consists of a set of triples of the form $\langle s, p, o\rangle$, where subject s denotes a global unique resource or a blank node; object o may denote either a unique resource, blank node or literal (i.e., string or number); and predicate p (a resource) denotes a relationship between s and o. RDF data can be represented as a directed labelled multigraph as defined below.

DEFINITION 1. *An **RDF data graph** $G_D = (V_D, E_D, L, \phi_D)$ is a directed labelled multigraph, where V_D is a set of vertices, E_D is a multiset of directed edges (i.e., ordered pairs of vertices). $(u, v) \in E$ denotes a directed edge from u to v. L is a set of available labels (i.e., predicates) for edges, and $\phi_D : E_D \rightarrow L$ is a labelling function.*

In RDF datasets, multiple triples may have the same subject and object and thus E_D is a multiset instead of a set. Also the size of E_D ($|E_D|$) represents the total number of triples in the RDF graph G_D. An RDF example in triplet form (TTL/N3 form) from Fig. 1 (G_D^i) is described below.

```
:H1 <has_appliance> :H1App1.
:H1 <near_by> :H2.
:H1App1 <power_usage> 50.
:H1App1 <status> ON.
```

Based on the above definition, we define an RDF graph event and RDF Graph streams as follows.

DEFINITION 2. *An **RDF Graph Event** denoted as (G_D, τ_i), consists of an RDF data graph G_D and a timestamp $\tau_i \in TS$, where TS is a set of totally ordered timestamps.*

DEFINITION 3. *An **RDF Graph stream**, denoted as $\mathcal{S}$, is a possible infinite set of RDF graph events.*

A conjunctive SPARQL query graph is used to match an RDF graph event within the RDF graph stream, and is represented by a set of triple patterns; it is defined as follows:

DEFINITION 4. *A **query graph** $G_Q = (V_Q, E_Q, L, \phi_Q, vars)$ is a labelled directed multi-graph, where V_Q is the set of vertices, E_Q is a multiset of edges connecting vertices in V_Q, L is the set of available labels for edges, $vars$ is a set of query variables, and ϕ_Q is the labelling function with $\phi_Q : E_Q \rightarrow L \cup vars$.*

A triple pattern tp is an RDF triple, where query variables ($vars$) are allowed in any position. The set of such triple patterns is called a basic graph pattern (query graph G_Q). Triple patterns are usually connected by the shared subjects or objects and a join occurs on their shared subjects or objects. In this paper, we only consider connected query graphs and we do not consider predicate joins, because variable predicates are not very common as shown in the previous study [3]; hence predicates of triple patterns are constants ($p \notin vars$). In the rest of the paper, we interchangeably use the terms triple pattern and edge of a query graph.

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

```
QUERY 1.    SELECT ?house, ?app, ?nbhouse WHERE  {
(tp1)       ?house <has_appliance> ?app.
(tp2)       ?app <status> ''ON''.
(tp3)       ?house <near_by> ?nbhouse.
(tp4)       ?nbhouse <id> 2.
}
```

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

Let $M(G_Q, G_D)$ be a function that generates the subgraph isomorphic matches of G_Q and G_D, then the result of the QUERY 1 over G_D^i in Fig. 1 is as follows: `(:H1, <has_appliance>, :H1App1), (:H1App1, <status>, ON), (:H1, <nearby>, :H2), (:H2, <id>, 2)`.

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

DEFINITION 5. *A **sliding window** $R_W(\tau) = W_x^\omega(\mathcal{S})$, which contains the slide x and window size ω ($x, \omega \in \mathbb{N}^+$), at each time τ converts the stream $\mathcal{S}$ into a set R_W containing recent RDF graphs from the stream $\mathcal{S}$, such that*

$$R_W(\tau) = \left\{ G_D \mid (G_D, \tau') \in \mathcal{S} \wedge \tau_b < \tau' < \tau_e \right\},$$

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

Windows are a central concept in stream processing: an application cannot store an infinite stream in its entirety. Instead windows are used to summarise the most recent set of elements [22] and evict the older ones from the system. For example `Range 5 Hours Slide 10 Minutes` describes a window of size `5 Hours` and a slide `10 Minutes` determines the granularity at which the window contents change.

Based on the above formal concepts, we present our problem statement as follows.

PROBLEM 1. ***Given** an RDF graph stream $\mathcal{S}$, a window $W_x^\omega(\mathcal{S})$, a query graph G_Q, and a matching function M, **compute** $M(Q, (G_D, \tau_i))$ i.e., subgraph isomorphism for an event $(G_D, \tau_i) \in \mathcal{S}$ at τ_i, such that*

$$M(Q, (G_D, \tau_i)) \oplus M(Q, R_W(\tau_j)) = M(Q, R_W(\tau_i)),$$

Algorithm 1 SPECTRA query processing: main process

```
1: for each event (G_D, τ_i) ∈ S do
2:   | views ← GRAPHSUMMARY(G_D, G_Q)                          ▷ Section 5
3:   | eventMatches ← QUERYPROC(views, G_Q, τ_i)              ▷ Section 6
4:   | if completeMatch(eventMatches, G_Q) then
5:   |   | prevMatches ← eventMatches
6:   | else
7:   |   | INCQUERYPROC(prevMatches, eventMatches, G_Q, τ_i)
8:   |   |                                                    ▷ Section 7
9:   | end if
10: end for
```

where $\tau_j < \tau_i$, and operator $\oplus$ incrementally uses the previously matched results $M(Q, R_W(\tau_j))$.

With loss of generality, in this paper we consider a time-based window. Other flavours, such as count-based windows, can easily be integrated into our model. In the rest of the paper, we simply use "event" and "stream" for RDF graph event and RDF graphs stream respectively.

4. SPECTRA FRAMEWORK: OVERVIEW

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

Algorithm 1 illustrates the global execution of the SPECTRA query processing. Each incoming event from a stream is first subjected to the graph summarisation process (*line 2*) (Section 5), where the query structure is utilised to prune "dangling" triples from each event before starting the matching process. The summarised events are materialised into a set of views using *bidirectional multimaps* to enable fast hash-joins. Then the system implements the joins between the views according to the triple patterns defined in the query graph (*line 3*). It incrementally constructs the indices between the joined triples as a by-product of the join process. If this process results into complete matches for a query graph, the timestamp of the event and incrementally produced indices are used to tag the matches in a timelist; and new matches are persisted into a set of *final views* (Section 6). Otherwise, the partial matches produced during the initial join process are processed with the previously computed matches (*line 7*) in the final views, while employing the incremental indexing. If this results into new matches, they are also persisted in the subsequent final views. This process ensures the completeness of the matches for a query graph for all the different types and sizes of events within a window (Section 7).

5. RDF GRAPH SUMMARISATION

Graph summarisation is the process of summarising a graph into a smaller graph that retains the useful characteristics of the original RDF data graph. That is, ignoring the part of the graph that contains no relevant triples with respect to the query. Thus, the query processing can be faster on summarised graphs than on the un-pruned ones [17]. Some RDF graph stores have extended bi-simulation and locality-based clustering approaches to perform join-ahead pruning

via graph summarisation [29, 35]. *Bi-simulation-based summaries* [29] are effective if only predicates are labelled with constants, such that multiple possible disconnected components of data graphs are merged into compact synopsis for indexing. It is an approximate solution and may contain errors. *Locality-based summaries* [35] use essentially graph clustering in which vertices of a data graph are partitioned such that vertices within each partition share more neighbours than nodes that are spread across the partitioning. Locality-based approaches are particularly effective if one or more of the subjects or objects in the query graph are labelled with constants.

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

DEFINITION 6. *A **query-based RDF summary graph** $G_S = (V_S, E_S, L, \phi_S)$ for a given query graph $G_Q(V_Q, E_Q, L, \phi_Q, vars)$ and an RDF graph $G_D(V_D, E_D, L, \phi_D)$ from an event at τ_i is a labelled, directed multigraph, where an edge $(u, v) \in E_D$ is in G_S iff,*

- *$\exists(u', v') \in E_Q$, such that, $\phi_Q((u', v')) = \phi_D((u, v))$,*
- *and if $u' \notin vars,\ u' = u$, and if $v' \notin vars,\ v' = v$.*

The set of triples within each summary graph G_S is stored using vertically partitioned tables, call *views*, each denoted as $\mathcal{V}_j$ with j a predicate label. An example is shown in Fig. 2(a): it is a summary graph produced using QUERY 1 for the RDF graph event G_D^i (in Fig. 1(a)). Fig. 2(b) shows a set of views, each containing a set of (s, o) pairs for a unique predicate, for the summary graph G_S^i.

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

The execution of the GRAPHSUMMARY operator is shown in Algorithm 2. It takes the input RDF graph of an event (G_D, τ_i), the registered query graph G_Q and a set of views, each for a $tp \in E_Q$. The algorithm performs the pruning and vertically partitioning of triples $t \in E_D$. First, we compare the predicate values for each triple pattern $tp \in E_Q$ to the predicates of triples $t \in E_D$ (*line 5*). Second, if the subject or/and object of the triple pattern contains a constant, we also compare it with the triples (s / o) in $t \in E_D$ (*line 6*). Finally, the matched set of triples for each triple pattern is encoded and stored in the corresponding view (*line 7*). By encoding, we mean encoding strings to numeric values by using a dictionary [9]; a routine process in RDF storage systems. This greatly compacts

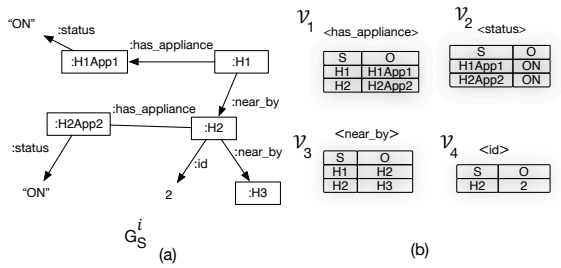

Figure 2 (a) Summary graph from the RDF Graph Event G_D^i using QUERY 1, (b) materialised views for the summary graph G_S^i.

Algorithm 2 RDF Graph Summarisation

```
1: viewset ← {V_tp1, V_tp2, ..., V_tp|EQ|}
2: procedure GraphSummary (G_Q, G_D)
3: for each triple pattern tp ∈ E_Q do
4:     for each triple t ∈ E_D do
5:         if pred(tp) = pred(t) then
6:             if (subj(tp) ∈ vars or (subj(tp) ∉ vars and subj(tp)
   = subj(t)) and (obj(tp) ∈ vars or (obj(tp) ∉ vars and obj(tp) =
   obj(t)))) then
7:                 V_tp ← V_tp ∪ {Encode(t)}
8:             end if
9:         end if
10:    end for
11: end for
```

the dataset representation and increases performance by performing arithmetic comparison instead of string comparison. Furthermore, dictionary encoding the blank nodes allows the matching process of each event in a manner consistence with the RDF data model [18].

6. CONTINUOUS QUERY PROCESSING

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

6.1 Incremental Indexing

The computation of *s-s* joins between a set of views is a straightforward procedure, and can be realised without the use of any indexing. However, complications arise when there are *s-o* / *o-s* joins between a set of views, i.e., for cyclic or tree-structured query graph patterns. Furthermore, for incremental pattern matching, the system requires to locate the correct part of the matches that are affected with the arrival of new events or with the eviction of old ones when the window slides. Generally, in static settings (as well as in RSP case) indices based on B+tree are used to locate the (s, o) pairs for multi-way join operations. These indices are discarded or built-up from scratch with new updates, thus, incurring delays during the rebuilding process.

To achieve both high performance and scalability, our index creation and maintenance solution is a byproduct of join executions between views, not of updates within a window. Thus, given a set of disjoint views only the joined triples are indexed using *sibling lists*; each for a unique view. Each sibling list is composed of the *sibling tuples* as defined below.

DEFINITION 7. *A **sibling tuple** is a 3-tuple $st = (\mathtt{id}(v), \mathtt{id}(u), g_i)$, where v and u are the (s, o) pairs joined on subject/objects between views,* `id` *is a function which assigns monotonically increasing numeric values to (s, o) pairs, called pair-ids. Finally, g_i is an ordi-*

nal number, which is assigned increasingly monotonically for each unique subgraph within the set of joined (s, o) pair.

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

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

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

Example 2. *Consider $\mathcal{V}_1$ and $\mathcal{V}_2$ in Fig. 3(a). The object column of the view $\mathcal{V}_1$ is joined with the subject column of $\mathcal{V}_2$, each distinct (s, o) pair within both views is assigned with a pair-id (Fig. 3(a)). The (s, o) pair* `(H1,H1App)` *identified with an id 1 in the view $\mathcal{V}_1$ has two joined (s, o) pairs in $\mathcal{V}_2$:* `(H1App,H1App1)` *and* `(H1App, H1App2)` *with ids 4 and 5 respectively. Thus, two sibling tuples st_1 and st_2 with the associated ordinal number* `00` *(only two bits ordinal numbers are used for the sake of presentation, we use 64 bits numbers for the implementation purposes) can be constructed as (*`4,1,00`*) and (*`5,1,00`*) (Fig. 3(b)). Similarly, join between the (s, o) pairs identified with 2 and 6 is indexed with sibling tuple st_3 (*`6,2,01`*); and the join between (s, o) pairs identified with 3, 7 and 8 is indexed with sibling tuples st_4 and st_5. The final matched result of such join operation is extracted by first collecting the distinct sibling tuples according to the ordinal numbers and then collecting all the (s, o) pairs associated to a distinct ordinal number. This procedure is described in the next section.*

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

Definition 8. *A **query conductor** is a data structure that first stores the materialised views from the summary graph (G_S) and then stores the joined (s, o) pairs evaluated from processing a query graph G_Q. It consists of three components:*

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

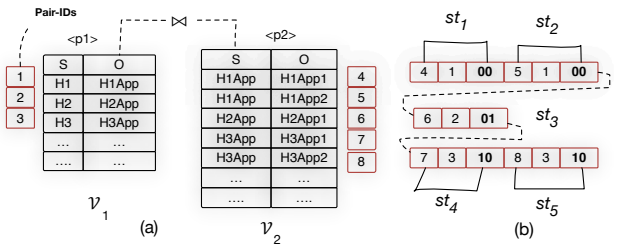

Figure 3 (a) Two views joined on an object and subject column, (b) sibling list constructed during the join operation for $\mathcal{V}_2$

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

(1) Multimaps are containers that associate values to the keys in a way that there is no limit on the number of same keys. It provides constant look-ups (considering there are no hash-collisions) for the keys (typically for s-s joins between two views). In order to provide constant look-ups for objects-related joins (i.e., s-o, o-s, o-o), by join operator reordering between two views, we extend the multimaps to bidirectional multimaps. That is, there is no limit on the number of same keys, and it also provides constant look-ups for the values.
(2) The sibling lists are used to implement the incremental indexing of matched (s, o) pairs in a set of views; it also assists in detecting the dropped (s, o) pairs with the expiration of sliding windows.
(3) The timelist stores the monotonically increasing timestamps of the events and provides a flat structure, since tree-like structures cannot cope with the frequent object insertion and deletion.

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

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

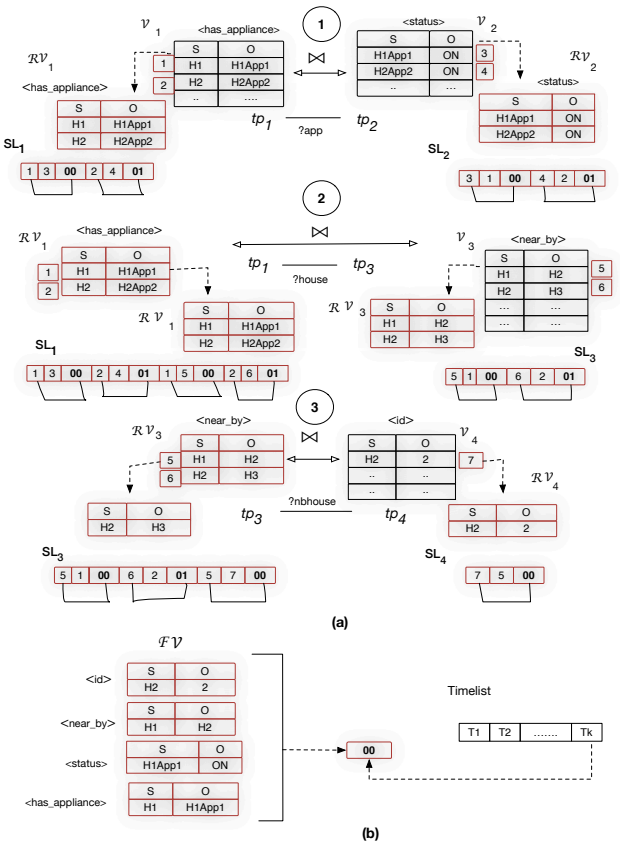

Figure 4 (a) Matching process of G_D^i with QUERY 1 as described in EXAMPLE 2, (b) a set of final views and a timelist

tuples are also assigned to a timestamp in the timelist (see Fig.4 (b)).

6.2 Query Processor

In this section, we provide the details of the query processor (QUERYPROC) operator from Algorithm 1 (*lines 3-5*). Its main objectives are as follows: (1) match an event with the query graph, if the process produces fully matched subgraphs then proceed towards the eviction process; (2) if the process produces only partially matched subgraphs within an event, then proceed towards the incremental query processor (Section 7).

QUERYPROC, as described in Algorithm 3, takes as input a query graph G_Q; the set of views for an event obtained from GRAPHSUMMARY operation; a set of final views ($\mathcal{FV}$) that contains complete matched results up to this point in a window; the timestamp of the current event τ_i; and the timelist τ_L with the window size ω and slide x. Note that, we follow the design principle of dynamic memory allocation in our system. Hence, we work with structurally-static query conductors, i.e., views, result-views, sibling lists etc., are initialised only once (see Algorithm 3, *lines 1-5*): memory is allocated at the creation or infrequently for resizing. Algorithm 3 first initialises the *impl-joins* set (*line 7*), which contains only the triple patterns whose views are not empty.

The main reason for this is that, due to the incremental nature of the algorithm there could be cases where only few views are updated and we have to incrementally join their results with previously computed ones (described later in Algorithm 4). Algorithm 3 then iterates over the $implJoin$ set. It first gets the joined triple

Algorithm 3 Query processing with QUERYPROC

1: $\tau_L \leftarrow timelist$
2: *viewset* $\leftarrow \{\mathcal{V}_{tp_1}, \mathcal{V}_{tp_2}, \ldots, \mathcal{V}_{tp_{|E_Q|}}\}$
3: *result-viewset* $\leftarrow \{\mathcal{RV}_{tp_1}, \ldots, \mathcal{RV}_{tp_{|E_Q|}}\}$
4: *sibling-set* $\leftarrow \{SL_{tp_1}, SL_{tp_2}, \ldots, SL_{tp_{|E_Q|}}\}$
5: $\mathcal{FV} \leftarrow \{\mathcal{FV}_1, \mathcal{FV}_2, \ldots, \mathcal{FV}_{|E_Q|}\}$
6: **procedure** QUERYPROC($G_Q, \tau_i, \mathcal{FV}$, *viewset*, ω, x, τ_L)
7: *impl-joins* $\leftarrow \{tp \in E_Q \mid \mathcal{V}_{tp} \neq \varnothing\}$
8: **for each** $tp \in$*impl-joins* **do**
9: | $tp_j \leftarrow$ `getJoinTP`(tp)
10: | `hashJoinAndIndex`($\mathcal{V}_{tp}, \mathcal{V}_{tp_j}, SL_{tp}, SL_{tp_j}, \mathcal{RV}_{tp}, \mathcal{RV}_{tp_j}$)
11: **end for**
12: **if** ($\forall tp \in$ *impl-joins*, $\mathcal{RV}_{tp} \neq \varnothing$ **and** |*impl-joins*|$= |E_Q|$) **then**
13: | $\mathcal{FV} \leftarrow$`extractMatches`($\mathcal{FV}$,*sibling-set*,*result-viewset*,τ_i, τ_L)
14: | `refresh`($\mathcal{FV}$,*sibling-set*, $\tau_i, \omega, x, \tau_L$) ▷ old events eviction
15: **else**
16: | Execute INCQUERYPROC ▷ Section 7
17: **end if**

pattern tp_j (*line 9*), which has to conduct the join operation with a certain tp. It then implements the hash-join and constructs the sibling lists for the two triple patterns by using the respective data structures. The `hashJoinAndIndex` function[6] (*line 10*) takes the viewset, result-viewset and sibling lists of the triple patterns to be joined. It either uses view ($\mathcal{V}$) or result-views ($\mathcal{RV}$) of the respective triple patterns and conducts the hash-join by iterating over the smallest one, i.e., employing the dynamic hash operator reordering, as described in EXAMPLE 3. Depending upon the type of join, it then fills the intermediate result-views, while incrementally updating each sibling list with sibling tuples.

The next phase of Algorithm 3 extracts the matched results according to the sibling tuples created during the join operations. It first checks if the join operations of $tp \in E_Q$ were successful (*line 12*). If so it initiates the `extractMatches` function (*line 13*). That is, first all the distinct ordinal numbers in the smallest sibling list are extracted. Second, a DFS on the sibling tuples is executed to extract all the matched subgraphs, while considering the sibling relationships between two joined triples (see EXAMPLES 2,3). Otherwise, it sends the partially matched results to the incremental query process. The eviction of the older triples in $\mathcal{FV}$ is performed with the `refresh` operation (*line 14*); its details are provided in the next section (Algorithm 4). Furthermore, the timelist τ_L is updated with the timestamp τ_i of the match event and its associated ordinal number during the execution of `extractMatches`.

Analysis. Each event arrival in the window triggers three main tasks: (1) implementing multi-way joins and indexing on the set of views produced from the GRAPHSUMMARY operation; (2) if all the joins produce results, matches are extracted using the sibling lists; and (3) deceased matches are removed from $\mathcal{FV}$. The cost of these operations can be described as follows. Operation (1) has two substeps, which includes linear hash-joins, and the construction of sibling lists, while propagating the ordinal numbers. Thus, if there are j join operations and l intermediate triples received for such joins, $m = l * 3$ is the size of each sibling list produced for indices. Then the total cost can be calculated as $\mathcal{O}(j(l\ log(m)))$ = $\mathcal{O}(j(l\ log(l)))$ for j join operations: $\mathcal{O}(log\ l)$ is the average cost of binary search through an sibling list. Operation (2) utilises traditional DFS to extract the matches using the sibling-set. Thus, if g is the number of ordinal numbers, and there are p distinct pair-ids in

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

the sibling tuples and c sibling tuples in the sibling lists, then the total cost of operation (2) can be described as $\mathcal{O}(g(p+c))$. Operation (3) consists of first collecting a set of timestamps that are outside the defined window and second deleting the (s, o) pairs associated to the timestamps. On average, its takes $\mathcal{O}(log\ k)$ for a binary search through the timelist to extract the deceased timestamps, and there will be $\mathcal{O}(d)$ final view access and deletion operations for d deceased triples. Regarding memory complexity, QUERYPROC obviously requires $\mathcal{O}(k)$, where k is the number of triples extracted during the execution of GRAPHSUMMARY and stored as matches in $\mathcal{FV}$. The question is finally how the number of triples k grows over time. Unfortunately, a respective formal analysis would require assumptions regarding the window size, data distribution itself and how it changes and matches to a particular query over time. Even under the assumption of a static data distribution, there is no general result.

Ordering of Join Operations. The joins between the sets of triple patterns $tp \in E_Q$ are commutative and associative [28]. An efficient join ordering results in smaller intermediate results leading to a lower cost of future join operations [27, 1]. Therefore, in Algorithm 3 and 4 (Section 7), we dynamically reorder the execution of join operations by considering the cardinality measures of each view after the graph summarisation process. Let us suppose that a multi-way join operation between views be $\mathcal{V}_1 \bowtie \mathcal{V}_2 \ldots V_{j-1} \bowtie \mathcal{V}_j$, where $|\mathcal{V}_i| \leq |\mathcal{V}_{i+1}|$ for every $i \in [0, j-1]$. Then the join sequence considering the left-deep evaluation strategy [24] is:

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

7. INCREMENTAL QUERY PROCESSING

This section provides the details of incremental query process INCQUERYPROC from Algorithm 1 (*line 7*). The main objectives of INCQUERYPROC are as follows: (1) computation of the partial matches, i.e., partially joined views are joined with the set of final views $\mathcal{FV}$; (2) the eviction of deceased matches from $\mathcal{FV}$, as it is not processed in QUERYPROC when INCQUERYPROC is executed.

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

Algorithm 4 then collects (in *incr-tplist*) all the triple patterns with non-empty intermediate result-views and iterates over it to conduct the joins with the set of final views $\mathcal{FV}$. It ignores all the other joins that were previously done during the execution of QUERYPROC. The remaining joins are conducted using $\mathcal{RV}_{tp}$ with the corresponding views in $\mathcal{FV}$ (*line 9*). If they all produce a non-empty intermediate result-view, the (s, o)-pairs are collected (same as in Algorithm 3) and added to the $\mathcal{FV}$ set with the timestamp of the event in τ_L (*lines 12-14*). Note that the analysis and cost of INCQUERYPROC operations can be directly inferred from QUERYPROC operations (Section 6.2).

EXAMPLE 4. *Consider the same* QUERY 1 *and the set of final views $\mathcal{FV}$ collected in* EXAMPLE 2*. Now consider the execution of* QUERY 1 *over G_D^j (see Fig. 1). As described in Fig. 5(a), two views $\mathcal{V}_1$ and $\mathcal{V}_2$ are materialised for triple patterns tp_1 and tp_2: data for other triple patterns are not present in G_D^j. Thus, the execution of* QUERYPROC *produces the partial matches (① in Fig. 5(a)), which have to be matched with $\mathcal{FV}$ to preserve the property of incremental evaluation. For such a task, we determine (i) the number of joins of tp_1 and tp_2, (ii) how many joins have already been executed after the* QUERYPROC *operations, and (iii) how many joins need to be further executed with $\mathcal{FV}$. As shown in* QUERY 1*, tp_1 has two joins: o-s join with tp_2 and s-s join with tp_3, while tp_2 has one s-o join with tp_1. During the execution of* QUERYPROC*, tp_1 has already computed the join with tp_2 using views $\mathcal{V}_1$ and $\mathcal{V}_2$; the same goes for tp_2, as shown in Fig. 5(a). Thus, we need to only implement the s-s join between $\mathcal{RV}_1$ of tp_1 and $\mathcal{FV}_2$ of tp_3. If the join produces a non-empty result-view then we add the partially matches from $\mathcal{RV}_1$ and $\mathcal{RV}_2$ in $\mathcal{FV}$. The whole process is shown in Fig. 5(a), where $\mathcal{RV}'_1$ is the result obtained by joining $\mathcal{FV}_2$ with $\mathcal{RV}_1$ ②. Finally, utilising the same technique in Algorithm 3, all the (s, o) pairs and their respective sibling relationships that are associated to the same ordinal numbers are added to the respective final views in $\mathcal{FV}$. Fig. 5(b) presents $\mathcal{FV}$ after the* INCQUERYPROC*, final views $\mathcal{FV}_3$ and $\mathcal{FV}_4$ are updated with new (s,o) pairs. As the partially joined results are added in these final views, we use a sibling relationship between the ordinal numbers* `(00,110)` *for the (s, o) pairs (*`H1App2,ON`*) and (*`H1,H1App2`*) in $\mathcal{FV}_3$ and $\mathcal{FV}_4$ respectively. Such sibling relationships between ordinal numbers are efficiently utilised for incremental updation of matches after the* `refresh` *operation (as described below).*

Algorithm 4 INCQUERYPROC

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

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

Incrementally Updating Query Matches. When a set of triples associated with an ordinal number and a timestamp is evicted from a window, all the existing matches in $\mathcal{FV}$ are updated accordingly.

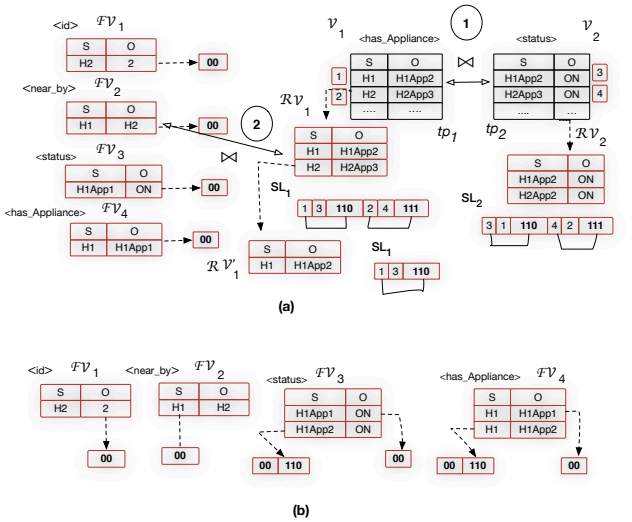

Figure 5 Incremental Processing of matched results of G_D^j in Fig. 1 with QUERY 1, as described in EXAMPLE 4.

Suppose that a triple/triple set associated with an ordinal number g_i in $\mathcal{FV}$ is evicted from the window, then there could be two cases: (1) $\texttt{sibling}(g_i) = \varnothing$, where sibling is a function, which determines all the sibling ordinal numbers of g_i, (2) $\texttt{sibling}(g_i) \neq \varnothing$. Case (1) describes that there are no matches effected by the removal of such ordinal number, as it does not have any siblings. Case (2) illustrates that there could be matches affected by the removal of g_i. Thus, the case (2) is incrementally handled as follows: if $\forall \texttt{sibling}(g_i), \exists (s, o) \in \mathcal{FV}_i$, $\forall i \in \{1, \ldots, |E_Q|\}$, then the existing matches are not affected by the remove operations. Otherwise, the siblings of partially matched (s, o) pairs are no longer within a window, and thus cannot be included in the output matches. Fig. 5(b) represents a set of final views, where the removal of triples associated with the ordinal number `00` would invalidate the partially matched results in $\mathcal{FV}_3$ and $\mathcal{FV}_4$. Thus, those triples cannot be included in the matched results output.

Semantics of Output Matches. Due to the incremental nature of our matching algorithm, the semantics of matches produced by a query at time τ_i for an event (G_i, τ_i) can easily be explained through an `Istream` operator from Continuous Query Language (CQL) [2]. That is, for each evaluation of QUERYPROC or INCQUERYPROC, only the newly found matches are reported in the output stream. This results in a less verbose output as compared to producing all the new and the old matches (that are already been reported) for each match execution.

8. EXPERIMENTAL EVALUATION

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

8.1 Experimental Setup

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

LUBM[7] is a widely used synthetic benchmark for benchmarking triple stores, and considers a university domain, with types like `UndergraduateStudent`, `Publication`, `GraduateCourse`, `AssistantProfessor`, to name a few. Using the LUBM generator, we create a dataset of more than 1 billion triples with 18 unique predicates. Concerning the queries, the LUBM benchmark has provided a list of queries. But many of these queries are simple 2-triple pattern queries or they are quite similar to each other. Hence we chose 7 representative queries out of this, as published in [4]; these queries range from simple star-shaped to complex cyclic patterns.

SEAS[8]project provides a real-world dataset[9] containing the power consumption statistics of family houses. It contains a set of power related attributes such as, measurement instrument types, voltages, watt values, etc. The dataset is available as RDF Data Cube and is mapped using the SEAS ontology[10]. The dataset is composed of power measurement values of a family house over the period of three years with a sampling period of 5 minutes. In total, it contains around 65 million triples. For queries, we generated one selective and one non-selective query from SEAS use cases; queries are illustrated in Appendix B.

Competitors. To compare against static triple stores, we chose two openly available and widely known systems: RDFox [26] and Jena [25]. Both of these systems are in-memory, and can be utilised for stream processing scenarios. For RSP systems, we selected CQELS [23] and C-SPARQL [8]. Both systems are widely used in the Semantic Web community and often compared in the literature even if both differ in their semantics: CQELS is based on push-based semantics and C-SPARQL is based on pull-based semantics. Our system resembles with CQELS due to its push-based semantics, i.e., queries are processed as soon as a new triple enters the system. In order to only compare the data structure and indexing technique of SPECTRA, we have also implemented a re-evaluation based version of SEPCTRA, i.e., only executing the QUERYPROC operator; persisting all the data in the views; and recomputing the joins for the whole window. It is denoted as S-Rev, while the general incremental version is denoted as S-Inc in the rest of the discussion.

Settings. The performance of query processing over sliding window depends on the window size ω; we intuitively expect it to spend more time on larger windows. For all the experiments (except for S-Inc-20), we use one triple for each event, the reasons are as follows: (1) CQELS and C-SPARQL only supports single triple streams;(2) this is the worst-behaviour in terms of query performance, as a large number of matching processes are executed, one for each triple. All the experiments were performed on an Intel Xeon E3 1246v3 processor with 8MB of L3 cache. The system is equipped with 32GB of main memory and a 256Go PCI Express SSD. It runs a 64-bit Linux 3.13.0 kernel with Oracle's JDK 8u05. For robustness, we performed 10 independent runs, and we report median time and memory consumptions. SPECTRA is implemented in Java and is openly available[11]

8.2 Evaluation

Relative Performance. The first question we investigate is *How does the incremental evaluation and indexing techniques perform as compared to the re-evaluation-based techniques?* This measures the performance gain while utilising SPECTRA. For this set of experiments, we use tumbling windows: when window size $\omega = x$ is equal to the slide granularity x, the sliding window degenerates to

[7] `http://swat.cse.lehigh.edu/projects/lubm/`

[8] `https://itea3.org/project/seas.html`

[9] `http://sites.ieee.org/psace-idma/data-sets/`

[10] `http://bit.ly/1UxxLXu`

[11] `http://spectrastreams.github.io/`

Table 1 Throughput analysis $\times 1000$ triples/second (rounded to the nearest 10) on LUMB dataset and queries over three different tumbling windows. Italics for incremental evaluation and best throughputs for re-evaluation are boldfaced. (•) indicates aborted execution

Windows	10 sec							50 sec							100 sec						
Queries	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q1	Q2	Q3	Q4	Q5	Q6	Q7
C-SAPRQL	12	54	8	27	76	88	19	9	36	6	19	44	58	11	•	•	•	12	20	22	•
Jena	35	80	15	68	150	110	26	12	53	9	37	108	73	15	•	11	•	15	33	24	•
CQELS	34	113	21	95	220	125	38	22	86	13	46	143	89	21	8	30	5	28	39	31	6
RDFox	28k	84	23	89	123	106	32	23	77	**17**	34	93	62	24	**12**	56	**9**	31	44	38	**15**
S-Rev	**60**	**750**	**46**	**356**	**846**	**796**	**64**	**36**	**322**	15	**88**	**637**	**748**	**28**	11	**80**	7	**56**	**161**	**179**	10
S-Inc	*81*	*848*	*54*	*425*	*980*	*925*	*73*	*52*	*567*	*26*	*180*	*876*	*853*	*50*	*30*	*241*	*18*	*146*	*540*	*415*	*27*

the tumbling window. The main reasons for using tumbling windows are as follows: (1) as the window does not slide incrementally, it can provide the measures of S-Inc overheads; (2) implementing sliding windows over existing in-memory stores is a complex task, thus can offer S-Inc an unfair advantage; (3) it can effectively provide a break-even analysis, where S-Inc outperforms S-Rev. Note that, to avoid any unfair advantage, we utilise the GRAPHSUMMARY operator on top of all the evaluated systems.

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

We start our analysis from the highly non-selective queries $Q1$, $Q3$ and $Q7$, each has a large number of triples associated with the triple patterns, and contains complex and cyclic patterns. Even with the GRAPHSUMMARY operator a large number of triples are inserted into the window, which results in higher query computation for each matching operation. Therefore, S-Rev for large windows performed slightly worse than RDFox; for each event, the matching process results in the reconstruction of indices. RDFox with its parallel, lock free architecture and one table indexing involves with few index updates, while CQELS requires substantial updates to its B+-tree indices for each triple in the stream. S-Inc on the other hand, move the matched triples to $\mathcal{FV}$ and the new events are only matched with the respective views in $\mathcal{FV}$, without re-constructing all the indices from scratch. C-SPARQL and Jena do not scale well with the increase in the size of the window, as their underlying storage structure (property tables) becomes quite dense for a large number of triples; C-SPARQL uses Jena and Esper[12] for its underlying execution model.

For selective queries $Q4, Q5$ and $Q6$, there are less triples in each window; most of the unrelated triples are pruned by our GRAPHSUMMARY operator. Query $Q5$ only contains 2 triple patterns and thus has even a smaller number of triples. Therefore, S-Rev performance on these queries is comparable to S-Inc, break-even analysis of both is provided later. From the rest of the systems, CQELS is a clear winner for smaller windows with its adaptive indexing technique and its Eddy operators provide an optimal query plans. However, its performance degrades with the increase in the number of triples within a window. RDFox performs better than Jena and C-SPARQL due to its parallel architecture. Query $Q2$ is less restrictive than the defined above; however, it only contains two triple patterns with one join. Thus, even with the increase in the number of triples all the systems perform better: the number of triple patterns in a query has a direct relation to the query performance.

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

[12] `http://www.espertech.com/esper/`

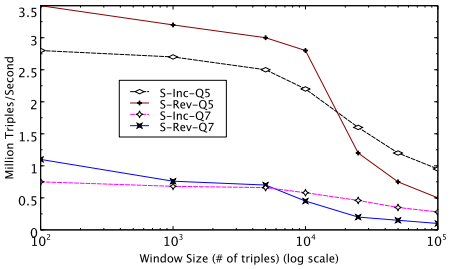

Figure 6 Break-Even Point for re-evaluation and incremental methods

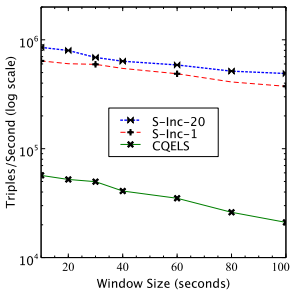

Figure 7 Performance of the non-selective SEAS $Q1$

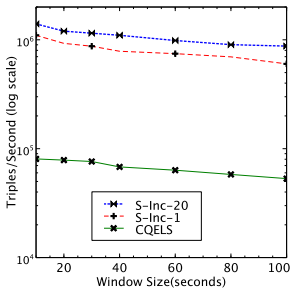

Figure 8 Performance of the selective SEAS $Q2$

Table 1 shows the comparative analysis of both strategies, however, due to the large size of the windows, S-Inc shows superior performance for all the queries. Fig. 6 shows the comparative analysis on relatively smaller window from 10^2 triples to 10^5 triples; for the sake of brevity we use the number of triples for the window size. We use the selective query $Q5$ and the non-selective complex query $Q7$ from the LUBM benchmark for this analysis. S-Rev performs less operations, thus the overhead of S-Inc is, as expected to be, higher than that of re-evaluation strategy. In most of the queries, S-Inc breaks-even for relatively small window sizes (between 10^4 and 10^5 for selective queries and between 10^3 and 10^4 for non-selective queries), conforming the particle utility of S-Inc on reasonable window sizes.

Fig. 6 shows that for $Q5$ S-Inc is about 20% lower on small window sizes and almost $3\times$ faster on large window. We are not concerned with slight slowdown, because at such small window sizes, extracting matches is unlikely to be the bottleneck of the application anyway. On the other hand, S-Inc yields large speed ups even at the moderate window sizes of 10^4 triples and we can observe even larger speed-ups when increasing the window size

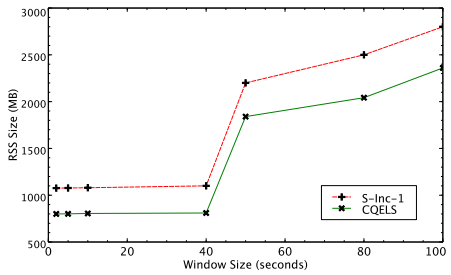

Figure 9 Resident set size (in MB) of S-Inc-1 and CQELS for SEAS $Q1$

further.

Sliding Windows. Next we investigate *How S-Inc performs when the window slides with variable granularity for triple and RDF graph streams?* This measures the performance of the system when it constantly resizes its window. For this set of experiments, we use a slide granularity of $x = 1$, i.e., each time the matching process fires, it handles the insertion and eviction of triples. This is the worst-case behaviour for sliding windows in terms of per-event cost. We use the SEAS dataset and its non-selective ($Q1$) and selective ($Q2$) queries for this set of experiments. Fig. 7 and 8 show the performance of $Q1$ and $Q2$ without using the GRAPHSUMMARY operator over CQELS engine; hence highlighting the importance of this operator. Note that we only use CQELS for comparative analysis, as C-SPARQL is much slower, as confirmed by our earlier experiment. For both $Q1$ and $Q2$ S-Inc-1, i.e., using an event containing 1 triple, is much faster than the CQELS engine, which performs re-evaluation, uses adaptive indexing and operator reordering.

As the window grows, S-Inc-1 is nearly an order of magnitude as fast as CQELS; the number of matches and triples in a window grows linearly (specially for $Q1$) and therefore the cost of scanning all the triples is quite high for the CQELS. Furthermore, it is very expensive to scan the large number of matches with the eviction of older triples. S-Inc, with the eviction of triples from the window, does not re-evaluate the query on the remaining triples; instead ordinal numbers and sibling relationships are utilised to determine the invalid matches. The performance of selective query $Q2$ (Fig. 8) shows the power of SUMMARGRAPH operators, where even with the increase in the size of the window, only the triple that conforms to the selective values of the query are added to the window.

Recall from Section 3, SPECTRA uses a general RDF graph model for the events, which allows a set triples to be enclosed in each event. The importance of this model, in terms of performance, is illustrated in Fig. 7 and 8, where S-Inc-20 uses events each of 20 triples. The total number of distinct matching operations for S-Inc-20 is less than S-Inc-1, where a new matching operation is for a batch of 20 triples, instead of 1 triple. This results in 25-35% increases in performance of the system for S-Inc-20 as compared to S-Inc-1. Furthermore, this also complies to the general streaming setting, as for relational model stream, each event consists of a set of attributes, which can be related to a set of triples for the attribute set in RDF graph streams.

Memory Consumption. *What is the effect of S-Inc data structure on the memory consumption?* Fig. 9 shows the resident set size (RSS; lower is better), which is measured using a separate process that polls the `/proc` Linux file system, once a second. Note that this method did not interfere with the overall timing results, from which we concluded that it did not perturb the experiments. Fig. 9 shows the comparative results of CQELS and S-Inc-1, and as expected, S-Inc takes slightly more memory than CQELS due to its bidirectional multimaps. However, this pays-off with the performance improvements. At small or moderate window sizes, the impact of window size is fairly minor on RSS as compared to base RSS of the entire process. Both systems continue to consume space linearly in size of the window.

9. CONCLUSION

This paper presents a framework that allows incremental indexing and evaluation over RDF graph streams, while utilising sliding window model. It uses a set of vertically partitioned views to collect the summarised data from each event and employs sibling lists to incrementally index the joined triples between views. The matched results are persisted in a set of final views, thus enabling the incremental evaluation with the arrival of new events. Given these properties, the experimental results shows that our approach clearly outperforms traditional offline/online indexing and re-evaluation-based solutions. Our future work includes extending the framework to integrate multiple simultaneous streams; out-of-order streams; incremental aggregate operators; and the state-based operators to enable complex event processing over RDF graph streams. Some of these extensions are discussed in Appendix A.

10. REFERENCES

[1] D. J. Abadi, A. Marcus, S. R. Madden, and K. Hollenbach. Sw-store: A vertically partitioned dbms for semantic web data management. *The VLDB Journal*, 18(2):385–406, Apr. 2009.

[2] A. Arasu, S. Babu, and J. Widom. The cql continuous query language: Semantic foundations and query execution. *The VLDB Journal*, 15:121–142, 2006.

[3] M. Arias and J. D. Fernández. An empirical study of real-world SPARQL queries. *CoRR*, abs/1103.5043, 2011.

[4] M. Atre and Chaoji. Matrix "bit" loaded: A scalable lightweight join query processor for RDF data. In *WWW*, pages 41–50, 2010.

[5] R. Avnur and J. M. Hellerstein. Eddies: Continuously adaptive query processing. In *SIGMOD*, pages 261–272, 2000.

[6] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *SIGMOD-SIGACT-SIGART*, pages 1–16, 2002.

[7] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *SIGMOD-PODS*, pages 1–16, 2002.

[8] D. F. Barbieri and Braga. C-SPARQL: Sparql for continuous querying. In *WWW*, pages 1061–1062, 2009.

[9] H. R. Bazoobandi, S. Rooij, F. Harmelen, and H. Bal. A compact in-memory dictionary for RDF data. In *ESWC*, pages 205–220, 2015.

[10] J. Broekstra and Kampman. Sesame: A generic architecture for storing and querying RDF and RDF schema. In *ISWC*, pages 54–68, 2002.

[11] J.-P. Calbimonte, O. Corcho, and A. J. G. Gray. Enabling ontology-based access to streaming data sources. In *ISWC*, pages 96–111, 2010.

[12] S. Chaudhuri and V. Narasayya. Self-tuning database systems: A decade of progress. In *VLDB*, pages 3–14, 2007.

[13] L. Chen and C. Wang. Continuous subgraph pattern search over certain and uncertain graph streams. In *IEEE Trans on Know. and Data Eng.*, pages 1093–1109, 2010.

[14] S. Choudhury, L. B. Holder, G. C. Jr., K. Agarwal, and J. Feo. A selectivity based approach to continuous pattern detection in streaming graphs. pages 157–168, 2015.

[15] W. Fan, J. Li, J. Luo, Z. Tan, X. Wang, and Y. Wu. Incremental graph pattern matching. In *SIGMOD*, pages 925–936, 2011.

[16] A. Gubichev and M. Then. Graph pattern matching: Do we have to reinvent the wheel? In *GRADES*, pages 8:1–8:7, 2014.

[17] S. Gurajada, S. Seufert, I. Miliaraki, and M. Theobald. Triad: A distributed shared-nothing rdf engine based on asynchronous message passing. In *SIGMOD*, pages 289–300, 2014.

[18] A. Hogan, M. Arenas, A. Mallea, and A. Polleres. Everything you always wanted to know about blank nodes. *Web Semantics: Science, Services and Agents on the World Wide Web*, 27–28:42 – 69, 2014.

[19] S. Idreos, M. L. Kersten, and S. Manegold. Database cracking. In *CIDR*, pages 68–78, 2007.

[20] S. Idreos, M. L. Kersten, and S. Manegold. Updating a cracked database. In *SIGMOD*, pages 413–424, 2007.

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

[22] J. Krämer and B. Seeger. Semantics and implementation of continuous sliding window queries over data streams. In *ACM Trans. Database Syst.*, volume 34, pages 4:1–4:49, 2009.

[23] D. Le-Phuoc, M. Dao-Tran, J. X. Parreira, and M. Hauswirth. A native and adaptive approach for unified processing of linked streams and linked data. In *ISWC*, pages 370–388. 2011.

[24] F. Liu and S. Blanas. Forecasting the cost of processing multi-join queries via hashing for main-memory databases. In *soCC*, pages 153–166, 2015.

[25] B. McBride. Jena: Implementing the RDF model and syntax specification. In *SemWeb*, pages 23–28, 2001.

[26] Y. Nenov, R. Piro, B. Motik, I. Horrocks, Z. Wu, and J. Banerjee. RDFox: A highly-scalable RDF store. In *Proc. of the 14th International Semantic Web Conference (ISWC 2015)*, Lecture Notes in Computer Science. Springer, 2015.

[27] T. Neumann and G. Weikum. The RDF-3X engine for scalable management of RDF data. In *VLDB*, pages 91–113, 2010.

[28] J. Pérez, M. Arenas, and C. Gutierrez. Semantics and complexity of SPARQL. In *ACM Transactions on Database Systems*, volume 34, pages 1–45, 2009.

[29] F. Picalausa, Y. Luo, G. H. L. Fletcher, J. Hidders, and S. Vansummeren. A structural approach to indexing triples. In *ESWC*, pages 406–421, 2012.

[30] K. Schnaitter, S. Abiteboul, T. Milo, and N. Polyzotis. Colt: Continuous on-line tuning. In *SIGMOD*, pages 793–795, 2006.

[31] U. Srivastava and J. Widom. Flexible time management in data stream systems. In *SIGMOD-SIGACT-SIGART*, PODS '04, pages 263–274, 2004.

[32] C. Weiss, P. Karras, and A. Bernstein. Hexastore: Sextuple indexing for semantic web data management. In *VLDB Endow.*, volume 1, pages 1008–1019, 2008.

[33] K. Wilkinson. Jena Property Table Implementation. In *SSWS*, 2006.

[34] D. Wood, M. Lanthaler, and R. Cyganiak. RDF 1.1 concepts and abstract syntax. In *W3C Recommendation, Technical Report*, 2014.

[35] L. Zou, M. T. Ozsu, L. Chen, X. Shen, R. Huang, and D. Zhao. gStore: a graph-based SPARQL query engine. In *VLDB*, pages 565–590, 2014.

APPENDIX

A. EXTENDING SPECTRA

As discussed in Section 7, SPECTRA supports the output of matches in an incremental manner. In this section, we discuss how the it can be extended to support the `Rstream` operator [2] to output all the matches that exist in a window at a certain time τ_i. Furthermore, we discuss the extension for the out-of-order stream model.

Extension of Output Semantics. The extension of the output model for SPECTRA is a straightforward procedure. As noted in Section 6 and 7, matched triples from each event are stored in a set of final views $\mathcal{FV}$. Thus, in order to extract all the matches $(M(Q, R_W(\tau_i))$ at each time τ_i for a query graph Q each match operation can visit all the available final views and extract incrementally stored triples. The results produced in this settings can be more verbose: the same matches can be present in the output that are computed at different evaluation times. Nevertheless, it can satisfy the semantics of `Rstream` operator from the CQL [2]: such semantics are used by the C-SPARQL engine for processing triple streams.

Out-of-order Streams. SPECTRA makes the general assumption that the events arrived within a streams are totally ordered. This assumption is not only considered by all the RDF stream processing systems, but also most of the DSMS complies to it. The total order assumption might not be met in practice because of network latencies and distributed data sources. Therefore, in case of out-of-order streams, the system can buffer the input events for a certain maximum amount of time and then reorder them [31].

B. SEAS QUERIES

Q1.

```
prefix seas: <http://purl.org/NET/seas#>
prefix m: <http://purl.org/NET/seas/measures#>
            SELECT *
            WHERE  { ?obs  m:V_RMS ?rm.
                     ?obs  m:V_THD ?thd.
                     ?obs   m:V_CF   ?cf.
                     ?obs   m:W   ?watt.
                     ?obs   m:Wh   ?watth.
                      ?obs   m:DPF   ?vah.
                    }
```

Q2.

```
prefix seas: <http://purl.org/NET/seas#>
prefix m: <http://purl.org/NET/seas/measures#>

            SELECT *
            WHERE  { ?obs  m:V_RMS ?rm.
                     ?obs  m:V_THD ?thd.
                     ?obs   m:V_CF   ?cf.
                     ?obs   m:W   ?watt.
                     ?obs   m:Wh   ?watth.
                      ?obs   m:DPF   ?vah.
                      Filter( ?watt > 10 && ?cf < 1.2)
                      }
```