DIONYSUS: Towards Query-aware Distributed Processing of RDF Graph Streams

Syed Gillani, Gauthier Picard, and Frédérique Laforest
Laboratoire Hubert Curien, UMR CNRS 5516, and Institute Henri Fayol, EMSE
Saint-Etienne, France
syed.gillani@univ-st-etienne.fr , gauthier.picard@emse.fr,
frederique.laforest@telecom-st-etienne.fr

ABSTRACT

Arguably, the most significant obstacle to handle the emerging application's data deluge is to design a system that addresses the challenges for big data's volume, velocity and variety. Work in RDF stream processing (RSP) systems partly addresses the challenge of variety by promoting the RDF model. However, challenges like volume, velocity are overlooked by existing approaches. These challenges demand optimised combination of scale-out and scale-up solutions. Furthermore, various other requirements for RSP systems, such as an efficient integration of distributed stream sources, storage of historical streams and their analysis, and integration of stateful operators to support complex event processing over streams are far from being addressed in an efficient way. Our vision is to design a general purpose RDF graph streaming system, which will be able to cope with distributed streams and shares local optimising strategies to allow different kinds of queries (analytical, streaming, sequence-based) through one query interface. The proposed system will offer a black-box solution that will allow analysts to tap in the goldmine of massive RDF graph streams. We consider the challenges and opportunities associated in designing such system, introduce our approaches to these topics, and discuss the components of our envisioned system.

1. INTRODUCTION

In a wide range of domains from social media to financial trading, there is a growing need of supporting continuous queries over predefined windows of data. The initial generation of stream processing systems (SPSs) were focused on delivering real-time response for the input that usually consisted of homogeneous stream tuples (relational) with predefined schemas. The implementation of such stream processing engines is relatively straightforward as long as the source data can be correctly represented in the data model of the SPS, and vice versa; albeit involving some manual work and inflexibility.

In order to provide a more flexible solution, and to enable reasoning capabilities for continuous queries; recently, the notion of RDF stream processing (RSP) systems has been introduced [7, 18]. The data for such systems are modelled as RDF graphs; providing the solution for variety and heterogeneity of data sources. RDF is a schema-free data model in which data are represented as subject-predicate-object ($\langle s, p, o \rangle$) statements called triples, and SPARQL is a standard query language for RDF with triple patterns as one of its main constituent. Today, RSP systems are turning into mature academic systems [7, 18, 10, 17], while broadening the spectrum of applications they serve. As a part of this process, we increasingly see the need to entertain these systems with functionality like scalability, distribution of sources, historical analysis on streamed data, and integration of new operators to take the advantages of the structured data model.

In SEAS [1] project, we tackle with a Smart Grid scenario, where a set of distributed and heterogeneous sources emanate data. The sources are mainly composed of sensors providing data about the power consumption at appliance level in each house, power generation/storage data, weather related data, usersâĂŹ activity data etc. Given the variety of data sources, the system must support an assortment of data sources, standard graph analytic (e.g., how many appliance are present in a house, their operational times and power ratings), complex analytic (e.g., finding areas in the city with power shortages and nearby power rich areas), real-time monitoring (e.g., detection of abnormalities in power usage, power disruptions), and complex event processing (CEP) (defining a sequence to trade electricity) over temporal and distributed events. The above mentioned scenario identify three main dimensions of optimisations and requirements to be included in the current RSP systems. That is, (i) distribution and scalability of RSP systems, (ii) enabling analytical queries over historical streams, and (iii) incremental evaluation to enable stateful operators for CEP.

First, as in classical data integration, continuous queries on data streams may involve combined processing over a number of streaming data sources physically distributed across the network. The currently available RSP systems require streams to be transmitted to a single location for centralised processing. Unfortunately, the continuous transmission of large number of rapid streams to a centralised locations may exceed the capacity of monitoring infrastructure. Thus, these approaches are not feasible for many real-world scenarios. Consider for instance, social network like Twitter, where on average there are around 6000 tweets per second and around 500 million tweets per day. An ad-hoc RSP system with a set of queries defined over long temporal windows (e.g., 1 day) can easily exhaust the available system resources, and requires a distributed computing model. Similar, for Smart Grid, each appliance in a house emanates on average 200K events per day; extending it for a set of appliances in a house and then for a set of houses would be beyond the capacity of existing RSP systems [13].

Second, the main optimisation goal of current RSP systems is to reduce the latency of results, since they mainly address what might

be called monitoring applications. That is, the systems were mainly not designed with storage in mind; while essentially all of the monitoring applications that we encountered had a need for archival storage. The storage of historical data within RSP systems would enable (i) analytical queries on the historical data, (ii) make predictive analysis; while comparing the historical and current data streams. Storing data from streams is not as simple as it sounds; the data from streams grow rapidly making it difficult to store and manage. Storage capacity, however is only one aspect of the problem. The data generated by streams can be so big that its analysis can also lead to severe time-bound issues. The complete analysis of stored data, however is rarely required and what users need instead is explorative access to the data to find interesting subsets that need further detailed and time-consuming analysis. For instance, a sensory device capturing the temperature of the room every 2 seconds would produce a lot of repetitive data. However a user might be only interested in temperature values for longer time periods with few distinct values.

Third, many real-time applications, not only require the on-line analytics of streams (as provided by all RSP engines), but also require stateful operators; which allows the new data elements to act as updates to the previously processed data elements. For instance, the data element describing the measurement of power consumption by an *appliance* represents an update to its previously measured values. This kind of scenarios is not uncommon in applications like Smart Grid, social networks, stock exchange, transport systems etc., where each new input might represents the updated value of an object. This means, as opposite to the *batch algorithms* – that recomputes new output from scratch by re-evaluating all the selected events – RSP system needs an incremental matching algorithm to minimise the computation by only evaluating the new events, while considering the already computed states of the previous ones. Furthermore, such stateful behaviour can also provides another interesting class of queries, i.e., sequence-based queries. These queries determine the defined stateful sequence of events and produce the output once it has been matched. These operators can be defined under the umbrella of complex event processing (CEP). The general RSP systems do not offer native support for CEP; thus, any solution built on top of them will suffer in terms of expressive power, usability, and performance. Few specialised systems [4] that support sequencing over an RDF model are defined in the literature; however, they lacks the scalability and distribution requirements as described earlier.

There is a rich literature on RSP system, CEP and distributed querying over RDF, but corresponding systems are rather rigid (i.e., can only analyse one triple at a time (for RSP and CEP), and distributed querying is based on static data), and do not deliver in terms of performance required for applications that we envisioned. Existing systems that are capable of providing some sort of functionalities as discussed above fall into one of the following categories: (1) distributed RDF batch processing systems that are optimised only for static data [28, 15], (2) centralised RSP systems that only provide real-time analytics [7, 18, 10, 17], (3) centralised CEP systems that only provide sequence operators [4]. Our ambitious goal is to develop a general-purpose system that not only provides all the required functionalities, but also shares optimisation strategies across the system; it will offer the abstractions, tools and dedicated algorithms needed for achieving these goals. The proposed system will make it feasible for analysts to use one query interface to post various different types of queries on distributed data streams.

We first present the state-of-the-art to draw the picture of the work required to achieve all our tasks. Second, we present a list of selected challenges. Third, we provide an overview of our envisioned approach. Fourth we provide various query optimisation techniques for our envisioned system.

2. STATE OF THE ART

The main object of the proposed research is to build a general purpose system which should support a wide variety of queries over RDF graph streams and on historical data. The existing techniques in this scenario can be divided into (i) RDF stream processing (RSP) systems, (ii) RDF storage systems for static data, (iii) CEP over RDF streams

[RDF Stream Processing] Stream processing of graph structured data (often dubbed as stream reasoning) – where streams are continuously processed together with semantics and rich background knowledge – has gained fair momentum. Existing techniques [7, 18, 10, 17] tackle diverse issues including, continuous query processing, stream reasoning, ontology maintenance, ontology-based data access. However, they are far too limited and complex to be applicable on gigantic distributed data streams. These solutions are usually optimised for centralised settings and cannot be directly adopted for federated/distributed settings. Furthermore, most of these techniques are based on the *triple stream model*, where each element/event within a stream is composed of a triple: a model directly inspired from the relational tuple stream model. For the same reason, most of these solutions (e.g., CQELS [18], C-SPARQL [7]) map RDF triples on relational tuples – to be handled by an underlying relational stream processor. We argue that this would not allow to reap the real power of structured events and it would be difficult to extend it for further graph operators; the W3C RSP[1] group took the same tone for triple stream model [2]. CEP over RDF [4] has the comparable story as that of RSP – with triple model as the dominant cause of controversy. We believe that an RDF graph model for streams would first enable to close the semantic gap between existing techniques, and second allows to tailor techniques from static RDF graph solutions to RDF graph streams.

[Centralised RDF Storage Systems] To cater the high volume of data, we need to store it in an efficient manner; this would enable near-to-real-time analytic queries and real-time streaming queries. Several techniques have been proposed to store graph structured data. These techniques range from native graph storage systems [29, 6, 27] to specialised RDF storage systems [5, 3, 23]. Most of these systems are optimised only for static data and they employ extensive indexing techniques to optimise query performance. For instance, RDF3x [3] builds several clustered B+-trees for all permutations of subjects, objects and predicates. Such extensive indexing techniques are not practical or feasible for high velocity graphs stream; as they would spend considerable amount of time and space for pre-processing and indexing RDF graph streams.

[Distributed RDF Storage Systems] In order to realise the distributed nature and amount of data on the Web, several techniques have been proposed for distributed RDF graph storage and querying. These techniques span from optimising a federated layer [24, 9, 21] on top of existing data stores to a dedicated solutions for clustering and querying RDF graphs in distributed manners [28, 15]. These techniques provide a good starting point but are not directly applicable for streaming settings, due to the following points. (1) Optimisation/distribution strategies are based on the assumption of static data, (2) the distribution/clustering of RDF graphs does not cater the heterogeneity of the sources; this can result into an increase in network traffic and replication of data in streaming settings. Furthermore, due to their re-evaluation model, such dis-

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

tributed approaches can lead to significant increase in query time with the increase of data size.

3. SELECTED RESEARCH CHALLENGES

In this section, we expose main challenges to be dealt for our envisioned system.

[Streams, Events and Query Model] One important problem for RDF graph-based stream processing is the lack of clean semantic models for defining streams and continuous queries to process them. There is no agreement even on the definition of basic terms such as "stream" and "event". Although most of existing systems are based on a *triple stream* model, it is one of the many possible points that shows the dependence of RSP engines on relational models. Hence, the existence of a wide variety of streaming applications not only introduces complexities in choosing the right engine for given applications, but also makes the application development and maintenance hard. The need of standardisation has been recently acknowledge by the W3C RSP working group and few of their initial propositions clear out the discussion toward the stream and event model. Furthermore, most of the existing RSP systems use extended forms of SPARQL to embed streaming operators. However, due to the lack of precise semantics, the query results for these RSP engines are not even comparable [12, 22].

[Handling Volume and Velocity under Distributed Settings] The problem of distributing data into a set of clusters, and then finding the query relevant data from all such distributed sources at federation layer has been dealt in static settings [24, 9, 21, 28, 15]. In such cases, data is known in advance and its distribution is performed once. However, this is not the case in streaming settings; data is not known in advance for distribution analysis and new sources are added dynamically and old sources provide data at variable velocities. Distributing and storing such a dynamic sea of information brings new challenges and thus needed to be dealt. Existing RSP systems are based on sliding window-based execution strategy: data in the windows are processed and expelled once a window is expired or it slides forward [22]. This first does not allow historical/analytical analysis on data. Second, the windows are devised in an ad hoc manner, thus are optimised only for small bandwidth.

[Incremental Indexing and Query Evaluation] The use of indexing (in distributed RDF settings) enables two primary tasks: (1) it determines the relevant distributed sources for a query, (2) it tunes the NP-hardness of subgraph isomorphism – that is matching a query graph and the data graph – into a more subtle problem; that is either joining the result of a set of triple patterns or exploring smaller subsets of query graph to join them for the final result [29]. Nevertheless, the good indexing strategies can prove to be quite significant for the performance of a system; we argue based on our experience with existing storage solutions that indexing itself can become a bottleneck to store and query RDF graph streams. For example, it takes more than 10 hours to build an optimised state-of-the-art index over a dataset of 2 billion triples on modern servers [26]. It is imperative therefore, to first develop techniques that interactively and adaptively build parts of the index, while focusing on the data necessary to answer queries and making the data available immediately. Second, the continuous streaming queries require incremental evaluation model. This means, the storage mechanism in this scenario must store the state of previously computed results and the new entries in the stream should act as an update to the previously stored states.

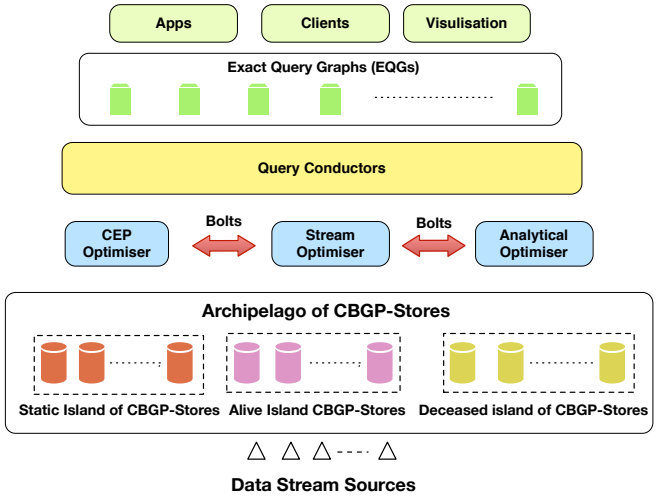

Figure 1: Layered Architecture of DIONYSUS

4. DIONYSUS

In this section, we describe the components of our envisioned system called as DIONYSUS[2]. Our system design is motivated by the following goals and is described in Fig. 1.

(1). We are interested in an efficient distribution of streaming data from a set of sources, which are not known in advance. Thus, our storage and data distribution model is envisioned by *Common Basic Graph Pattern store* (CBGP-store). Each CBGP-store is assigned with a generic BGP (i.e., a set of triple patterns) generated automatically/manually from the domain ontology and domain use cases (see Section 4.1). A collection of such stores exposes (i) fresh incrementally computed results of a set of CBGP for streaming queries, (ii) the set of previously computed results for off-line analytical queries. As we see in Section 4.1, this enables the use of light-weight incremental indexing technique to efficiently store data for each CBGP.

(2). The second goal of our approach is to push the intensive query optimisation and processing locally at each CBGP-store for *Exact Query Graphs* (EQGs) registered by a user. An EQG is a more selective form of CBGP and it contains (i) subsets of triple patterns that are distributed among CBGP-stores, (ii) SPARQL 1.1 operators, such as `select, optional, union, filter, group by`, etc. The results of each EQGs are accumulated at the federated level; Section 4.2 describes such techniques.

(3). Our third goal is to enable different kinds of queries – such as analytical, streaming, sequence-based – through a single query interface. A query interface encompasses subsets of CBGP-stores that can be abstracted as *islands of CBGP-stores* (see Fig. 1). This would enable to share query results computation and local optimisation strategies. For example, users would like to get the result of (i) an analytical query describing the number of active appliances and their power usage in a house, and the result of (ii) a sequence-based query to determine the sequence of power usage by appliances. This calls for a single query that can be optimised according to its defined operators as presented in Section 5.

(4). Our fourth aim would be to provide the *semantic completeness* and *locations transparency*. That is, a new source can be added without affecting the integrity constraints, and a user query can

[2]**D**istr**I**buted a**N**al**Y**tical, **S**treaming and **S**equence q**U**erie**S**

span multiple islands of CBGP-stores. This would enable to first, share the optimising strategies defined for each CBGP-store, second to reduce the network traffic by employing local optimisation and computation strategies.

In summary, our envisioned system can provide a way not to drown in the sea of information emanating from heterogeneous distributed sources. It filters unnecessary information, which otherwise can result in excessive use of storage and computational resources. The framework is designed to minimise the burden of query evaluation at the federation layer and to share local optimisation strategies across the islands of CBGP-stores.

4.1 Common Basic Graph Pattern Stores

The main difference between the static and streaming data distribution is its availability. The static data is available beforehand for analysis and thus various techniques such as semantic hash-based pertaining [19] can be utilised to partition it into a set of clusters. However, the streaming data can not be available in advance for distribution analysis. Thus, we envisioned an ontology-based data distribution. Inspired by works on ontology partitioning [25, 20], ontology modularisation [11] and ontology segmentation [8], we propose to generate a set of common basic graph patterns (CBGP) by analysing the structural relationships described in the domain ontology/ontologies. Given a set of ontologies $O = \{o_1, o_2, \ldots, o_n\}$ used by a set of streams, a function $F(O)$ produces a set of CBGP by analysing the structural relationships within ontologies and the general use cases defined for a particular domain. Intuitively, we can say that each CBGP should contain information about a coherent subtopic within an ontology. The concepts within each CBGP are semantically connected to each other and should not have strong dependencies with the information outside the CBGP. Thus, each CBGP = (C, D) contains a set of concepts C and links D between them represent different kinds of dependencies; where D can be reflected in the definition of an ontology or can be implied by the intuitive understanding of the concepts and background knowledge about the respective domain. Generally, CBGPs consist of different types: star-shaped, shallow tree shaped, deep tree shaped, graph with loops etc. Each CBGP is mapped to a data store; thus forming a set of dataset clusters compose of data aggregated from a set of sources. Note that such data distribution may results in data duplication across the set of CBGP-stores. However the following benefits would outweigh such shortcomings.

(1) Aggregating commonly linked concepts within a single CBGP-store. This would results in (i) querying processing to be focussed at local levels, and (ii) reducing the network traffic and load at federation level. The data duplication among CBGP-stores ensure that each query answer can be computed locally.

(2) Surviving in the sea of information by only filtering and storing the relevant information. The CBGP-stores can act as data filters, where only the relevant data based on the selected graph relationships will be stored and the full dataset from a set of sources is summarised by a set of CBGPs. The summarised data acts as a surrogate for the original data and are queried instead of the complete dataset.

(3) In-memory incremental evaluation of the CBGPs; storing only the states of the computed results, rather than all the data elements from a set of sources.

(4) It enables the query-aware light-weight and adaptive indexing technique for storing CBGPs results. As described earlier, implementing an incremental and adaptive indexing is one of the major challenges in storing and querying dynamic streams of RDF graphs.

A collection of CBGP-stores is abstracted under an island, where each island is assigned with a set of query conductors. Query conductors shares optimisation strategies through bolts (see Fig. 1). Each CBGP-store is divided into three flavours: static, alive or deceased. This classification is based on the fact that a streaming query requires the current incrementally computed results, while the analytical or predictive queries base their execution on historical states of incrementally computed results. The static CBGP-stores are used to enrich the streams with the static background knowledge – one of the main property of RSP engines. Each CBGP-store captures a fraction of data from different sources, thus the storage of alive CBGP-store and query computation can be done completely in the main-memory, while the deceased and static CBGP-stores can utilise disk-based storage.

4.2 Query Computation via Query Conductors

The query conductor layer is the working horse of the system. This layer is responsible for conducting various high-level optimisations and accumulating results of the user-defined queries on an archipelago of CBGP-stores (see Fig. 1). When a customised user query – called as an Exact Query Graph (EQG) – is registered, the query conductor first determines its type (analytical, streaming or sequence-based query) and creates an abstract syntax tree (AST). It then divides the EQG into a set of subquery graphs, alongwith their temporal information, in case of streaming and sequence-based queries. It next utilises the information of its islands of CBGP-stores and orchestrates the execution of each subquery graph on a set of CBGP-stores. The query conductor itself does little computation other than concatenating or temporal sorting of results from CBGP-stores. This design permits us to take the advantages of local optimisation strategies implemented for each CBGP-store. Note that EQGs are more specialised and selective versions of CBGP, thus the set of triple patterns within each EQG can be easily decomposed and registered to CBGP-stores.

Results of streaming and sequence-based EQGs are held in main-memory buffers, and they are updated whenever there is a newly computed incremental results in the corresponding alive CBGP-stores. The query conductor also ensures that the data movement is not too expensive; we will explore the tactics for moving results based on operators involved, processing activity at various CBGP-stores, and overall network activity. That is, if a certain EQG results in an increase network traffic or query processing at federation level, we can create a new CBGP-store by analysing the structure of the EQGs.

5. QUERY OPTIMISATION

In this section, we discuss various opportunities offered by our envisioned system to optimise the performance of different types of queries: analytical, streaming and sequenced-based.

5.1 Analytical Queries

Analytical queries for historical data analysis usually contains multiple aggregation phases. For example, the query to find the average power consumption by each house grouped by area is described in QUERY 1 (using SPARQL[3] syntax). It contains three specifications: (i) graph pattern matching to compute the query-related

[3]http://www.w3.org/TR/sparql11-query/

subgraphs corresponding to the power consumption of a house and area information, (ii) grouping the resulting patterns based on the values of area-house combinations, and (iii) aggregating the values of the power consumption to compute the average. Now consider that the house and its power consumption values lies in a CBGP-store D_1, while information regarding the area lies in another CBGP-store D_2, and there can be n CBGP-stores.

QUERY 1. *Analytical query for Smart Grid use case*

```
SELECT ?area, ?house, AVG(?power)    (iii)

 WHERE
{
?house :location ?l.
?house :powerSource ?source.          (i)
?source :value ?power.

?l :partOf ?area.
?area :name ?areaName.
}
GROUP BY (?area)                     (ii)
```

Traditionally in federated settings, the two basic optimisation techniques to compute such a query graph are: (i) compute each triple pattern in a query graph against all the available data stores and the results are joined at the server or (ii) evaluating each triple pattern in a nested loop join (NLJ) fashion; that is by substituting the results obtained from one triple pattern into another. These techniques and their optimised version [24] performs poorly for highly selective and complex analytical queries. [16] provides various optimisation strategies to cater analytical queries for federated static data sources and can act as a good starting point to extend it for dynamic settings, albeit in a different way. Since previous approaches assumed a cost-based model, they would not be efficient to support the addition of new sources and the generation of new CBGP-stores. The reason is that, they would require to maintain a model of each subquery graph operation and the resources it needs. That is, essentially the query optimiser need to understands all the operations and storage techniques in all the distributed resources, while continuously updating it for dynamic streams.

At opposite, we envisioned a black box approach, where no information about the local query optimisations is known at the server level. Each subquery graph of EQGs is sent to the corresponding CBGP-store, where a local optimiser determines the subquery join operations. Using the same example as described above, we can compute the graph pattern matching – which is typically join intensive – and aggregation of values at D_1 store. Similarly, the basic graph pattern matching for area-related query graph patterns will be performed at D_2 store. The results of both of these local processes are sent back to the analytical query optimiser, which uses the cardinality of the results from D_1 and D_2 to efficiently order the joins between house and area results, and finally performs the grouping. Our black-box approach also allows the semantic completeness and location transparency; new sources and CBGP-stores can be easily updated without remodelling/recalculating local optimisation strategies.

5.2 Continuous Streaming Queries

Continuous streaming queries are typically registered on a set of sources and the computation of the results are bounded by a window – which slides by certain elements count or time interval. For example, see QUERY 2 (using CQELS [18] syntax). It determines the subquery graph for power consumption on a source s_1 and the weather related subquery graph on source s_2, within a window of 2 hours that slides every 2 second. The results of the subquery graphs are joined on a common variable `?l`.

The scalability of continuous streaming queries are case dependent; there are two main flavours discussed in the literature. First, scaling a large number of queries by distributing their execution. Second, scaling a complex query that needs large working memory and might not fit within a single machine. The first case is easy to handle; a shared nothing execution architecture – i.e., neither streams or memory storage is shared among processors – can be utilised to run multiple instances of the streaming engine, each running a subset of the queries. However, scaling complex queries is still an open issue.

QUERY 2. *Streaming query for Smart-Grid use case*

```
SELECT ?power, ?house, ?temp, ?Wspeed, ?hum
WINDOW 2 HOURS
 WHERE
{
STREAM <http://example.org/powersource> [Range 2s]
{
?house :location ?l.
?house :powerSource ?source.
?source :value ?power.
}
STREAM <http://example.org/weathersource> [Range 2s]
{
?l :temperature ?temp.
?l :windSpeed ?Wspeed.
?l :humidity ?hum.
}

}
```

We argue that our system design could handle both cases efficiently with the following points. (1) We can utilise multiple query conductors to serve a set of EQGs, where query conductors can easily be distributed into a set of machines. (2) The main processing of a EQG is performed in a distributed manner on a set of CBGP-stores. That is, a EQG is parsed into a set of subquery graphs, each is assigned to a CBGP-stores. The query execution results are stored in the distributed cache and then joined on common variables (i.e., `?l` in QUERY 2) to produce the final results; the static CBGP-stores further provide the functionality of joining the static background knowledge with the streams. Such query execution strategy enables to share the results of common query graph patterns within a set of EQGs. Furthermore, there is also the case of incremental evaluation of the query result. Traditionally, QUERY 2 is executed in a re-evaluation manner [22, 18]. That is, the results for each subquery graph are indexed (usually B+ tree) and for each new event, (i) the result of each subquery graph is re-evaluated, (ii) the join on the common variable is re-evaluated, and (iii) all the computation is performed in an ad-hoc manner. Therefore, existing approaches are unable to support the scalability, performance and incremental evaluation requirements posed by the streaming queries.

5.3 Sequence-based Queries

The distribution of sequence-based queries is an open research issue and there seems to be no effort in the context of the semantic web community. A sequence-based query determines a data sequence, which in our case is an ordered sequence of RDF graphs. Formally, $S = \{(G^1, t_1), (G^2, t_2), \ldots (G^n, t_n)\}$ is defined as a sequence of n RDF graphs, where each graph contains a set of triples and a timestamp t_i in which the recording was made. Given a set of subquery graphs (QG) and an ordering/sequencing function $O : QG \rightarrow \mathbb{N}$, we determine the ordering of RDF graphs matched to the subquery graphs. A key observation in this case is that we need to execute or assemble the results of each subquery graphs in a way that it follows the ordering defined for QGs. For example, consider QUERY 3 (using IntellCEP syntax [14]), it determines the sequence

of RDF graph events (SEQ(A,B)), where the power consumption of a house is greater than a certain threshold value, followed by specific weather conditions.

Existing approaches [4] for executing such queries have various shortcomings; (i) they utilise a triple-based model, where only one triple is permitted in each event, (ii) they utilise ad hoc settings, where expensive indexing is used for ordering functions, and (iii) they are based on a single stream model. These shortcoming makes them unsuitable for many real-world use cases, where distribution is the key to cater huge volumes of data.

QUERY 3. *Sequence-based query for Smart Grid use case*

```
 SELECT ?house,?l,?power
 WITHIN 24 hours
 PARTITION BY (?house)
 FROM STREAM S1 <http://example.org/powersource>
 FROM STREAM S2 <http://example.org/weathersource>
 WHERE
{
SEQ (A, B)

 A ON S1
{
?house :location ?l.
?house :powerSource ?source.
?source :value ?power.
FILTER (?power > 50)
 }

 B ON S2
{

?l :temperature ?temp.
?l :windSpeed ?Wspeed.
?l :humidity ?hum.
FILTER (?temp > 20 && ?Wspeed > 10)

  }


}
```

Our approach of distributed CBGP-stores can however be useful in this scenario. Based on our earlier optimisation techniques, we can send each subquery graph to the corresponding alive CBGP-stores. The computation of graph pattern matching and aggregation on subquery graph is computed locally, while the results along-with their temporal properties can be utilised to determine the sequence defined on a set of events. That is, the query evaluation is broken into several steps in a pipeline that matches the events with a subquery graph and republish the matched events to a step further in the pipeline. The sequence of events within a pipeline can be matched by either utilising a rule-based system or an automata-based approach. Crucially, this step requires deep insight into the temporal measurements and state management for aggregate operators.

6. CONCLUSION

Even though the scalability, state management and distribution of sources are very common requirements for RSP system, there is currently no system that can provide a generic solution to accommodate these attributes. In this paper, we summarised the challenges and opportunities provided by a distributed general purpose system for RDF graph stream processing. We propose DIONYSUS that will provide one query interface to enable analytical, streaming and sequence-based queries. It will support islands of data stores each assigned to a CBGP; thus filtering and indexing the RDF graphs in an incremental manner. Furthermore, the use of such system allows to share optimisation strategies and query matches within a set of queries; leading to the desired scalability requirement presented by real-world systems.

7. REFERENCES

[1] https://itea3.org/project/seas.html.

[2] https://www.w3.org/community/rsp/wiki/RDF_Stream_Models. Priminilary draft Link "https://goo.gl/iLQscV".

[3] T. N. 0001 and G. Weikum. Rdf-3x: a risc-style engine for rdf. *PVLDB*, 1(1):647–659, 2008.

[4] D. Anicic and Fodor. EP-SPARQL: a unified language for event processing and stream reasoning. In *WWW*, 2011.

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

[6] M. Atre and J. A. Hendler. BitMat: A main memory bit-matrix of rdf triples. In *In: Proceedings of the 5th International Workshop on Scalable Semantic Web Knowledge Base Systems.*, 2009.

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

[8] M. Bhatt, C. Wouters, and Flahive. Semantic completeness in sub-ontology extraction using distributed methods. In *ICCSA*. 2004.

[9] C. Buil-Aranda, M. Arenas, O. Corcho, and A. Polleres. Federating queries in {SPARQL} 1.1: Syntax, semantics and evaluation. *Web Semantics: Science, Services and Agents on the World Wide Web*, 18(1):1 – 17, 2013. Special Section on the Semantic and Social Web.

[10] J.-P. Calbimonte, O. Corcho, and A. J. G. Gray. Enabling ontology-based access to streaming data sources. In *ISWC'10*.

[11] M. d'Aquin, M. Sabou, and E. Motta. Modularization: a key for the dynamic selection of relevant knowledge components. In *1st International Workshop on Modular Ontologies, WoMO'06*, 2006.

[12] D. DellâĂŹAglio, J.-P. Calbimonte, and Balduini. On correctness in rdf stream processor benchmarking. In *The Semantic Web âĂŞ ISWC 2013*. Springer Berlin Heidelberg, 2013.

[13] R. C. Fernandez, M. Weidlich, P. Pietzuch, and A. Gal. Scalable stateful stream processing for smart grids. In *DEBS*, pages 276–281, New York, NY, USA. ACM.

[14] S. Gillani, G. Picard, F. Laforest, and A. Zimmermann. Towards Efficient Semantically Enriched Complex Event Processing and Pattern Matching. In *OrdRing 2014 @ ISWC 2014*, Trentino, Italy, Oct. 2014.

[15] S. Gurajada, S. Seufert, I. Miliaraki, and M. Theobald. TriAD: A distributed shared-nothing rdf engine based on asynchronous message passing. In *SIGMOD*, 2014.

[16] D. Ibragimov, K. Hose, T. B. Pedersen, and E. Zimányi. Processing aggregate queries in a federation of SPARQL endpoints. In *ESWC*, pages 269–285, 2015.

[17] S. Komazec and D. Cerri. Sparkwave: Continuous schema-enhanced pattern matching over rdf data streams. In *DEBS*, 2012.

[18] D. Le-Phuoc and Dao-Tran. A native and adaptive approach for unified processing of linked streams and linked data. In *ISWC*. 2011.

[19] K. Lee and L. Liu. Scaling queries over big rdf graphs with semantic hash partitioning. *Proc. VLDB Endow.*, 2013.

[20] B. MacCartney, S. McIlraith, and Amir. Practical partition-based theorem proving for large knowledge bases. In *IJCAI*, 2003.

[21] K. Makris, N. Bikakis, N. Gioldasis, and S. Christodoulakis. SPARQL-RW: Transparent query access over mapped rdf data sources. In *EDBT*, pages 610–613, New York, NY, USA, 2012. ACM.

[22] A. Margara, J. Urbani, F. van Harmelen, and H. Bal. Streaming the web: Reasoning over dynamic data. *Web Semantics: Science, Services and Agents on the World Wide Web*, 25(0):24 – 44, 2014.

[23] Y. Nenov, R. Piro, B. Motik, I. Horrocks, Z. Wu, and J. Banerjee. RDFox: A highly-scalable RDF store. In *ISWC*. Springer, 2015.

[24] A. Schwarte, P. Haase, and M. Schmidt. FedX: A federation layer for distributed query processing on linked open data. In *ESWC'13*.

[25] H. Stuckenschmidt and M. Klein. Structure-based partitioning of large concept hierarchies. In *The Semantic Web âĂŞ ISWC 2004*, volume 3298, pages 289–303. Springer Berlin Heidelberg, 2004.

[26] Z. Sun, H. Wang, H. Wang, B. Shao, and J. Li. Efficient subgraph matching on billion node graphs. *Proc. VLDB Endow.*, 2012.

[27] P. Yuan, P. Liu, B. Wu, H. Jin, W. Zhang, and L. Liu. Triplebit: A fast and compact system for large scale rdf data. *VLDB'13*.

[28] K. Zeng, J. Yang, H. Wang, B. Shao, and Z. Wang. A distributed graph engine for web scale rdf data. In *PVLDB'13*.

[29] L. Zou and M. Tamer. gStore: a graph-based SPARQL query engine. *VLDB J.*, pages 565–590, 2014.