Vitor Pinheiro de Almeida

DSCEP: An Infrastructure for Decentralized Semantic Complex Event Processing

Tese de Doutorado

Thesis presented to the Programa de Pós–graduação em Informática of PUC-Rio in partial fulfillment of the requirements for the degree of Doutor em Ciências–Informática.

Advisor: Prof. Markus Endler

Rio de Janeiro
September 2021

Vitor Pinheiro de Almeida

DSCEP: An Infrastructure for Decentralized Semantic Complex Event Processing

Thesis presented to the Programa de Pós–graduação em Informática of PUC-Rio in partial fulfillment of the requirements for the degree of Doutor em Ciências–Informática. Approved by the Examination Committee:

Rio de Janeiro, September 17th, 2021

Vitor Pinheiro de Almeida

Graduated in computer science by the Pontifícia Universidade Católica do Rio de Janeiro.

Bibliographic data

Almeida,Vitor Pinheiro de

DSCEP: An Infrastructure for Decentralized Semantic Complex Event Processing / Vitor Pinheiro de Almeida; advisor: Markus Endler. – 2021.

86 f: il. color. ; 30 cm

Tese (doutorado) - Pontifícia Universidade Católica do Rio de Janeiro, Departamento de Informática, 2021.

Inclui bibliografia

1. Informática – Teses. 2. CEP Semântico. 3. Processamento de Eventos Complexos. 4. Stream Reasoning. 5. Processador de Stream RDF. 6. Infraestrutura Distribuída. I. Endler, Markus. II. Pontifícia Universidade Católica do Rio de Janeiro. Departamento de Informática. III. Título.

CDD: 004

I dedicate this thesis to my parents and grandmother
and especially to my wife, she is the person who made possible the completion
of this work.

Acknowledgments

To my wife Roberta, who was by my side every time during the development of this thesis. She is the person who made possible the completion of this work.

To my parents for their emotional and financial support and endless encouragement.

To my grandmother, who is both an inspiration and a gift to me.

To my advisor Professor Markus Endler for the stimulus and partnership to carry out this work.

To Professor Edward Hermann, Sukanya and Professor Kurt for the various inputs and constructive conversations about this thesis.

To CNPq and PUC-Rio, for the aids granted, without which this work does not could have been accomplished.

This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior – Brasil (CAPES) – Finance Code 001.

Abstract

Almeida,Vitor Pinheiro de; Endler, Markus (Advisor). **DSCEP: An Infrastructure for Decentralized Semantic Complex Event Processing**. Rio de Janeiro, 2021. 86p. Tese de Doutorado – Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Many applications require the processing of event streams from different sources in combination with large amounts of background knowledge. Semantic CEP is a paradigm explicitly designed for that. It extends complex event processing (CEP) with RDF support and uses a network of operators to process RDF streams combined with RDF knowledge bases. Another popular class of systems designed for a similar purpose is the RDF stream processors (RSPs). These are systems that extend SPARQL (the RDF query language) with stream processing capabilities. Semantic CEP and RSPs have similar purposes but focus on different things. The former focuses on scalability and distributed processing, while the latter tends to focus on the intricacies of RDF stream processing per se. In this thesis, we propose the use of RSP engines as building blocks for Semantic CEP. We present an infrastructure, called DSCEP, that allows the encapsulation of existing RSP engines into CEP-like operators so that these can be seamlessly interconnected in a distributed, decentralized operator network. DSCEP handles the hurdles of such interconnection, such as reliable communication, stream aggregation and slicing, event identification and time-stamping, etc., allowing users to concentrate on the queries. We also discuss how DSCEP can be used to speed up monolithic SPARQL queries; by splitting them into parallel subqueries that can be executed by the operator network or even by splitting the input stream into multiple operators with the same query running in parallel. Additionally, we evaluate the impact of the knowledge base on the processing time of SPARQL continuous queries.

Keywords

Semantic CEP; Complex Event Processing; Stream Reasoning; RDF Stream Processor; Distributed Infrastructure.

Resumo

Almeida,Vitor Pinheiro de; Endler, Markus. **DSCEP: Uma Infraestrutura Distribuída para Processamento de Eventos Complexos Semânticos**. Rio de Janeiro, 2021. 86p. Tese de Doutorado – Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Muitas aplicações necessitam do processamento de eventos de streeams de fontes diferentes em combinação com grandes quantidades de dados de bases de conhecimento. CEP Semântico é um paradigma especificamente designado para isso, ele extende o processamento complexo de eventos (CEP) para adicionar o suporte para a linguagem RDF e utiliza uma rede de operadores para processar streams RDF em combinação com bases de conhecimento em RDF. Outra classe popular de sistemas projetados para um proposito similar são os processadores de stream RDF (RSPs). Estes são sistemas que extendem a linguagem SPARQL (a linguaguem de query padrão para RDF) para adicionar a capacidade de fazer queries em stream. CEP Semântico e RSPs possuem propositos similares porém focam em objetivos diferentes. O CEP Semântico, foca na scalabilidade e processamento distribuido enquanto os RSPs focam nos desafios do processamento de streams RDF. Nesta tese, propomos o uso de RSPs como unidades para processamento de streams RDF dentro do contexto de CEP Semântico. Apresentamos uma infraestrutura, chamada DSCEP, que permite o encapsulamento de RSPs existentes em operadores do estilo CEP, de maneira que estes RSPs possam ser interconectados formando uma rede de operadores distribuída e descentralizada. DSCEP lida com os desafios e obstáculos desta interconexão, como comunicação confiável, divisão e agregação de streams, identificação de eventos e time-stamping, etc., permitindo que os usuários se concentrem nas consultas. Também discutimos nesta tese como o DSCEP pode ser usado para diminuir o tempo de processamento de consultas SPARQL monolíticas, seja dividindo-as em subconsultas e operando-as em paralelo através do uso de operadores ou seja dividingo a stream de entrada em multiplos operadores que possuem a mesma query e são executados em paralelo. Além disso também é avaliado o impacto que a base de conhecimento possui no tempo de processamento de queires contínuas.

Palavras-chave

CEP Semântico; Processamento de Eventos Complexos; Stream Reasoning; Processador de Stream RDF; Infraestrutura Distribuída.

Table of contents

List of figures

List of tables

Por vezes sentimos que aquilo que fazemos não é senão uma gota de água no mar. Mas o mar seria menor se lhe faltasse uma gota.

Madre Teresa de Calcutá, *1910-1997*.

1
Introduction

Semantic complex event processing (Semantic CEP) is a form of event stream processing that attempts to combine the generality and inference capabilities of RDF and related Semantic Web technologies with the efficiency and scalability of complex event processing (CEP).

In CEP [5, 6] a network of operators is used to search for patterns in event streams. An event is a piece of structured data with a fixed interpretation. Each operator in the CEP network takes (primitive) events as input, looks for a given pattern, and produces new (complex) events as output if any matches are found.

An example application of CEP is in the smart city scenario, where people can be notified about when their buses will arrive at the bus station. The events, in this case, are related to the user and to the buses containing a timestamp, user location, bus location, and time for the bus to reach different locations. For instance, a user may be interested to *estimate how many minutes the bus of line A will take to reach the user's location.* [7].

Contrast this with Semantic CEP. In Semantic CEP [8, 9, 10, 11], the simple, structured events of traditional CEP are replaced by sets of RDF triples (or RDF graphs), the event streams become RDF streams, and the search patterns become RDF graph patterns. More fundamentally, in Semantic CEP, there is the possibility of combining in the search pattern large amounts of background knowledge stored in knowledge bases (KBs). This, together with the ability to make inferences, enabled by ontologies and rules, opens up many possibilities. For example, instead of considering only the user and bus line locations to estimate when the next bus will arrive, the user might ask to *consider only buses that will not cross any street in which a dangerous event has happened in the past 30 minutes.* [10]. Listing 1.1 shows all data used to answer the question; although the user and bus location pattern is still there (line 2, 3, and 4), the other part of this query can only be solved by accessing and combining background knowledge (lines 7, 8, 9 and 10) in meaningful, semantically aware ways [12].

Listing 1.1: Smart city scenario example use case.

```
1 Event Stream:
```

```
{(UserSilva, hasPosition, LocationA)}
{(BusLine10, hasPosition, LocationB)
 (BusLine10, hasEstimateTimeToReachLocationA, 8mins)}

Knowledge Base:
{(BusLine10, hasLocationInPath, LocationC)
 (BusLine10, hasEstimateTimeToReachLocationC, 20mins)
 (LocationC, hasDangerousEvent, DangerEventA)
 (DangerEventA, hasOccurred, InLessThen30mins)}
```

A research effort closely related to Semantic CEP is Stream Reasoning (also called Linked Stream Data [13] or Semantic Streams). Stream Reasoning aims to enable the semantic processing of RDF streams in combination with background knowledge. What distinguishes Stream Reasoning from Semantic CEP is that Stream Reasoning tends to focus less on efficiency and scalability. Most Stream Reasoning solutions extend SPARQL (the RDF query language) to operate over RDF streams. These solutions, which we will call collectively *RDF Stream Processors* (RSPs; [14]), include Streaming SPARQL [15], C-SPARQL [16], $\text{SPARQL}_{\text{Stream}}$ [17], EP-SPARQL [18], CQELS [19], and TEF-SPARQL [20]. These are all standalone processing engines. They were designed for main memory processing and offer little or no support for distributed, decentralized processing.

This last remark is particularly important. Although there are other solutions built specifically for Semantic CEP (solutions which do not rely on RSP engines, such as SPA_{SEQ} [21]) these are neither as mature nor as popular as the RSP engines, especially within the Semantic Web community.

One drawback of the RSP engines is that there is no uniform standard or a set of requirements to follow to design and develop them. As a result, most RSP engines do not need to generate an output ready to be consumed as input by another RSP engine. Also, RSP engines do not need to offer support to multiple input streams and are not required to provide window operators. These engines differ concerning scalability, expressiveness, reasoning capabilities, and the query language supported. Most RSP engines do not support RDF graph streams, which are essential for enabling some use cases in which it is impossible to represent an event using only a single RDF triple. RDF graphs are a set of RDF triples that represents a single piece of information; these basic concepts will be explained in detail later on in Chapter 2.

1.1 Motivational Use Case for Semantic CEP

Social-media analytics attempts to extract valuable information from social media content. This information includes trending topics, real-time ratings, and emotions, and can used to derive correlations between events which at first might seem unrelated. This kind of analysis is essential to activities such as decision-making and risk assessment and can be done in real-time or over historic data. We argue here that Semantic CEP is better suited for this kind of application than traditional RSP-based stream processing.

Consider an application that attempts to use Twitter and Tumblr[1] posts to correlate positive comments about a musical artist with television shows. More specifically, we want to find posts that say something positive about a musical artist while also mentioning in the same post a television show. This query, call it Q, can help us identify, for instance, which television shows are good publicity for a given artist.

First, note that we need to analyze two RDF streams simultaneously (Twitter and Tumblr). So, we need the ability of processing multiple streams, which is a basic requirement of Semantic CEP (but not supported by most RSPs).

We assume that the two RDF streams were preprocessed and contain triples resulting from sentiment analysis. If posts are represented as RDF graphs then query Q can be implemented as a single query that operates over a stream of posts (graphs). The support for a graph-based event model is another basic requirement of Semantic CEP, and this ability greatly simplifies our job: as we can be sure that the triples of a same graph will not be separated between windows (when they are processed as batch windows). Again, some RSP do not support a graph-based event model.

Finally, Semantic CEP provides access to a background KB as a basic requirement (which is not the case for RSPs). This access is essential to answer query Q, which uses the KB for determining whether a given resource is a musical artist or a television show.

1.2 Research Questions and Contributions

Before this thesis, in past works of our research group, we were evaluating with a more hands-on focus approach to the use of stream reasoning and how the processing time behaves depending on the size and complexity of the background knowledge base [22]. Our team concluded that reasoning using the

[1]Tumblr: `https://www.tumblr.com`

data stream with a background knowledge base is costly. Additionally, it was clear that knowledge bases and RDF streams could enrich some use cases by using the ability to make inferences enabled by ontologies and rules. Moreover, the use of RDF is widely used in the literature for data integration purposes. The work of Reis et al. [22] contributed to formulating part of the following research questions and also is one of the motivations of this thesis.

The following are the research questions of the thesis:

RQ1 Is it possible to connect reasoning (using ontologies and logic rules) with CEP? By doing so, we intent to reason with data both on the stream and on the KB.

RQ2 Is it possible to connect different RSP engines to make them work on the same operator network?

RQ3 Is it possible to make an RSP engine work as a SCEP engine? Maybe if we develop an infrastructure able to work with different RSP engines, we can approximate the RSP engine current model to the processing model of Semantic CEP.

RQ4 What is the impact of including the knowledge base in the stream processing in query processing time?

RQ5 What is the gain, in terms of query processing time, of dividing a monolithic query into multiple smaller queries to be executed in an operator network?

With this in mind, and with the goal of approximating the current RSP engines to the processing model of Semantic CEP, we propose DSCEP [23]: a software infrastructure for decentralized semantic complex event processing. The DSCEP infrastructure allows the encapsulation of RSP engines into CEP-like operators and permits these operators to be interconnected in a distributed, decentralized operator network. The hurdles involved in such interconnection (reliable communication, stream aggregation and slicing, event identification and time-stamping, etc.) are handled by the DSCEP infrastructure allowing the user to concentrate on the queries. Also, DSCEP may be used as a middleware service for semantic processing on ContextNet (see [24]) as discussed on the example above.

One interesting application of DSCEP, which we investigate in this thesis, is the parallelization of monolithic RSP queries. In RDF stream processing, the most expensive parts of queries are often those parts that access the external KB. This problem is exacerbated when the KB is large, which is not uncommon.

DSCEP can be used to speed up such monolithic RSP queries by either breaking them into subqueries that run in parallel and access only the relevant part of the KB or by splitting the windows of the input stream to process them in parallel.

The contributions of this thesis are the following:

1. A system model for decentralized Semantic CEP which enables the construction of distributed Semantic CEP operator networks using existing RSP engines.

2. An implementation of this model, called DSCEP, developed on top of Apache Kafka (a distributed commit log which efficiently stores and delivers data streams).

3. An implementation of an adaptation of a type of stream parallelization, executed inside the operator, called Data-Parallel CEP [25]; which can boost the overall performance and scalability of the infrastructure.

4. An experimental evaluation of DSCEP using a social-media analytics scenario, which uses both C-SPARQL [16] and a custom implementation of an RDF stream processor developed by this thesis as an RSP engine. The experiments measure the speed up made possible by DSCEP when: (a) breaking a monolithic query into a network of operators with each operator accessing only a part of the original KB, and; (b) splitting the windows of the input data stream and processing them in parallel.

2
Background

2.1
RDF, Reasoning, and Knowledge Bases

Resource Description Framework (RDF) [26] is a graph-based framework for stating facts about resources. In RDF, a fact is represented as a triple (s, p, o) consisting of a subject s, a predicate p, and an object o. When we assert an RDF triple we are saying informally that the relationship indicated by the predicate holds between the resource denoted by the subject and the resource or value denoted by the object [26].

In RDF, predicates and resources are defined using URIs (or, more generally, IRIs). There are plenty of URIs with predefined meanings available on the Web. These are called *vocabularies* and can be used to describe things in different domains.

Here is an RDF document (in Turtle syntax [27]) describing a person:

```
1 @prefix ex: <http://ex.org/> .
2 @prefix foaf: <http://xmlns.com/foaf/0.1/> .
3 ex:Bob a foaf:Person .
4 ex:Bob foaf:name "Bob Brickley" .
```

The prefix declarations (lines 1 and 2) establish that `ex` and `foaf` will be used as abbreviations for the corresponding URIs. The next two lines describe one (s, p, o) triple each. The first triple (line 3) states that the resource `http://ex.org/Bob` is a person (`http://xmlns.com/foaf/0.1/Person`) in the sense defined by FOAF[1], a vocabulary of people-related terms. And the second triple (line 4) states that this person is called (again in the sense defined by FOAF) "Bob Brickley".

From this example, we can see that the "no fixed schema" nature of RDF gives it a great flexibility, especially for representing and connecting pieces of knowledge. For instance, it is straightforward to add to the previous document more triples which further describe `http://ex.org/Bob` using FOAF or other vocabularies, or to use this resource as the target of relationships with other persons or things.

[1]FOAF: `http://www.foaf-project.org/`

Another important feature of RDF is its well-defined semantics [28]. RDF semantics comes with an associated notion of entailment which enables the derivation of new triples from previously stated triples (an ability which is sometimes referred to as *reasoning*). The reasoning process in RDF is governed by the use of user-defined theories of domain knowledge, called ontologies. An *ontology* is essentially a set of statements describing the concepts and relations of a particular domain. Ontologies are expressed in ontology languages, such as RDFS [29] and OWL [30, 31], and are encoded in RDF. For our purposes, a *knowledge base* (KB) is a set of statements (triples) possibly describing ontologies.

The statements in an ontology can be partitioned into two groups: TBox and ABox. The TBox contains general statements that apply to all individuals, such as “every person is an animal”, “every animal is either dead or alive”, etc., while the ABox contains statements about specific individuals, such as “Bob is a person”. Reasoning is the process of validating the ABox with the TBox and of making implicit statements explicit. So, for example, from the TBox statements “every person is an animal” and “every animal is either dead or alive”, we can infer “every person is either dead or alive”. This combined with the ABox statement “Bob is a person”, gives us “Bob is either dead or alive”, a previously implicit statement which now can be used in further inferences with other TBox and ABox statements.

2.2 RDF Streams

In Semantic CEP and Stream Reasoning, we deal with not only with static RDF documents but also with RDF streams. An RDF stream is a stream of (s, p, o) triples where each triple is associated with a timestamp. RDF streams are commonly used for building streams of knowledge graphs, facilitating the connection between the data on the stream and the knowledge base by using interlinked and interoperable information structures. For instance, RDF streams can be used on the data flow from social media input from all its users or even to represent semantic sensor data gathered from sensors in IoT applications.

There are two ways of identifying events in RDF streams. The first way is to consider each triple in the stream a distinct event—each triple is treated as a self-contained unit which can be interpreted independently of the other RDF triples in the stream.

Consider the following RDF stream:

Listing 2.1: An example of an RDF triple stream.

```
(ex:Bob,ex:tweetAbout,dbp:Demi_Lovato)     1000
```

```
2 (ex:Kirk,ex:tweetAbout,dbp:Democracy)      2000
3 (ex:John,ex:tweetAbout,dbp:Ketchup)        3000
```

Here each line represents an (s, p, o) triple with an associated timestamp in milliseconds. The prefix `dbp` stands for `http://dbpedia.org/resource/` (DBpedia is a knowledge base extracted from Wikipedia). The first triple (line 1) states that Bob (`ex:Bob`) posted a tweet about Demi Lovato (`dbp:Demi_Lovato`) and that this occurred with a time offset of 1000ms. The second triple (line 2) states Kirk tweeted about Democracy at 2000ms, and the third triple (line 3) states that John tweeted about Ketchup at 3000ms. Each of these events expresses a complete fact which can be interpreted independently of the other events in the stream.

The second way of identifying events in an RDF stream is to consider as distinct events not the individual RDF triples but whole RDF graphs. That is, the RDF stream is treated as a stream of RDF graphs. An RDF graph [32] is a set of RDF triples, where the vertices are the subjects and objects and the edges labeled by properties and connect subjects with objects. Additionally, RDF graphs can contain blank nodes which are vertices without any information itself and they can be either a subject or an object vertice. Blank nodes are often used to describe multi-component structures, like RDF containers, putting them together by connecting to the same blank node.

Listing 2.2 is an example of a single event in an RDF graph stream:

Listing 2.2: A single event in an RDF graph stream.

```
1 (_:t8bfc2f68,rdf:type,sioc:post)             1000
2 (_:t8bfc2f68,dc:created,
3   "2013-02-02T02:32:56"^^xsd:dateTime)       1000
4 (_:t8bfc2f68,sioc:id,"8bfca2f68")            1000
5 (_:t8bfc2f68,schema:mentions,_:e8bfc68_1)    1000
6 (_:e8bfc68_1,rdf:type,nee:Entity)            1000
7 (_:e8bfc68_1,nee:detectedAs,"ed sheeran")    1000
8 (_:e8bfc68_1,nee:hasMatchedURI,dbp:Ed_Sheeran) 1000
```

These eight triples form an RDF graph which represents a single tweet. The vertices `_:t8bfc2f68` and `_:e8bfc68_1` are blank nodes, used to group parts of information about a tweet. Each triple conveys only part of the information in the tweet. The first triple (line 1) states that the resource being described (`_:t8bfc2f68`) is a post (`sioc:post`), the second triple (lines 2–3) states that this resource was created on February 2nd, 2013, at 02:32:56, and so on. Note that the timestamps of the triples are all equal (1000ms). The rationale for this is that the temporal order of the triples within the graph is irrelevant. Timestamps are only meaningful for events, which in this case consist of whole graphs.

2.3 RDF Stream Processing

RDF Stream Processing (RSP) [14, 33] engines are SPARQL-based systems for processing RDF streams. SPARQL is the query language of RDF. Most RSP engines extend the SPARQL language with specialized constructs for continuous queries over RDF streams [34]. As examples of RSP engines we can cite Streaming SPARQL [15], C-SPARQL [16], SPARQL$_{\text{Stream}}$ [17], EP-SPARQL [18], CQELS [19], and TEF-SPARQL [20].

Here is a query in the SPARQL dialect of C-SPARQL [16]:

```
1 REGISTER QUERY TweetStream AS
2 SELECT { ?user ?gender }
3 FROM STREAM <http://ex.org/Tweets> [RANGE TRIPLES 3]
4 FROM <file:///Person-KB.rdf>
5 WHERE  { ?user ex:tweetAbout ?x .
6          ?x rdf:type foaf:Person .
7          ?x foaf:gender ?gender. }
```

This query operates over a stream of triples describing tweets (lines 1 and 3). The stream is assumed to be in the same format as the first RDF stream listed in Listing 4.1, i.e., each triple determines a new event. The query continuously reads windows of three triples from the stream (line 3), extends each window with the triples in the external KB (line 4; which in this case is a KB describing persons) and selects from each extended window all pairs (*user*, *gender*) where *user* tweeted about a person with *gender* (lines 5–7). That is, it selects all `?user` and `?gender` such that there is in the extended window:

(i) a triple relating a subject `?user` via predicate `ex:tweeted` to some resource `?x`;

(ii) a triple relating `?x` via `rdf:type` to `foaf:Person`;

(iii) a triple relating `?x` via `foaf:gender` to `?gender`.

If we apply this query to the RDF stream listed in Listing 4.1 we will get the pair: (`ex:Bob`, `"male"`). The above query uses count-based windows (it slices the stream into windows of a fixed size).

2.3.1 Limitations of current RSP engines

Most RSP engines support count-based windows, which divide the stream based on a specific number of RDF triples or RDF graphs; time-based windows, which use the timestamp of events to slice the stream and; sliding windows which can be applied both to count-based or time-based windows. What most of these engines do not support are streams of RDF graphs, such as the second RDF stream listed in Listing 2.2.

In Chapter 3, we will see that DSCEP, the Semantic CEP infrastructure we propose, can be used to overcome this and other limitations of RSPs. In particular, DSCEP handles window slicing and event delimitation so that windows and events are specified using the chosen RSP engine query language and published into the DSCEP infrastructure to enable another RSP engine to read and process them. DSCEP also handles input stream aggregation since some RSPs cannot process more than one input stream at a time.

Two other drawbacks of RSPs which are addressed by DSCEP are their poor compositionality and their lack of support for partitioning the external KB. These two problems are somewhat related. By poor compositionality we mean that it is hard to use the results of a query as input for another query; and it is even harder if the two queries run on different RSPs—an issue that arises from the lack of standardization surrounding the topic of RDF streams [35]. In C-SPARQL, for instance, every successful query triggers a user callback which is the sole responsible for handling the query results. As we will see, DSCEP provides a clean interface for connecting operators (RSPs) with an appropriate implementation for the result callback in the case of C-SPARQL.

The lack of support for KB partitioning in RSPs is related to the problem of query compositionality. If one can decompose a complex query into more basic, elementary queries, then one should also be able (in some cases) to decompose a large KB into smaller KBs to be used by each of the smaller queries. This issue is particularly troublesome in C-SPARQL. To combine the triples within the current window with the triples from the KB, the C-SPARQL engine simply adds all triples of the KB to the window, producing an "extended window". If the KB is too large this method becomes impractical. We will return to the topic of KB partitioning later.

Regarding the RSP engine, there are some key issues to be addressed. One of them is the availability of these RDF streams. There is no uniform standard to follow in order to design a RSP engine. Current standards apply to RDF data exchange and are produced by the Semantic Web's community[2][3]. RSP engines normally benefit with that by being able to use these standards, but so far, there are only few adoptions. As a result, most RSP engines do not need to generate an output ready to be consumed as input by another RSP engine.

Moreover, RSP engines do not need to offer support for multiple input streams and are not required to provide window operators. These engines differ concerning scalability, expressiveness, reasoning capabilities, and the query language supported. For instance, CSPARQL [16] and CQELS [19] support count-based and time-based windows over RDF triple streams while EP-SPARQL [18] supports only temporal operators as part of the query language. These differences make it challenging to make them work together by connecting them.

Although one of the advantages of the RDF data model is to enable reasoning, RSP engines are not required to provide full reasoning capabilities. For example, some RSP engines do not allow the use of a background knowledge base, limiting the reasoning only for the query level [3]. Others even define a subset of the SPARQL language to be allowed to avoid certain SPARQL operations that are difficult to accomplish in a scalable way.

2.4 Semantic CEP

Semantic CEP [8, 9, 10, 11] attempts to combine semantic technologies (RDF, ontologies, KBs, etc.) with complex event processing. In CEP [5] events streams are processed while they traverse an operator network. CEP engines, different from RSP engines, are commonly used in a distributed way and are often highly parallelized [25].

Ideally, a Semantic CEP solution should preserve the fundamental characteristics of the CEP paradigm while being compatible with existing RSP and other stream reasoning solutions. This brings a set of requirements which can be summarized as follows [21]:

(i) Streams are sequences of RDF triples annotated with a timestamp, with observed events being described either by individual triples or by whole graphs;

(ii) It must be possible to combine RDF streams with background knowledge residing in external KBs;

[2]W3C - Semantic Web: `https://www.w3.org/standards/semanticweb/`
[3]RSP Group: `https://www.w3.org/community/rsp/`

(iii) Multiple RDF streams can be processed at once using different stream aggregation and window partitioning strategies;

(iv) Processing can be done in a compositional manner (i.e., the output of an operator should be ready to be used as input for another operator in the network).

Note that the requirement (iv) listed above does not imply that SCEP engines must provide a distributed infrastructure to connect multiple operators. It simply says that the output generated by the SCEP engine should be a dataset represented in RDF and timestamped, which is ready to be consumed by another SCEP instance.

In traditional CEP, one can also combine data from the stream with data residing on external databases, figure 2.1 illustrates this process. It shows that in order to build a pattern which combines data from the stream with an external database, there must exist two queries, one with a pattern to match the data on the stream, and another with a pattern to match the data on the database. That's because it is only possible to query the database during the callback, which is triggered when the events in the window match the pattern expressed in the continuous query. Another problem of doing this type of query using CEP is that the stream and the database can be represented using different languages and schemas. Most of the cases it is required to write two queries using two different languages, one to find a pattern on the stream and another to find a pattern on the external database.

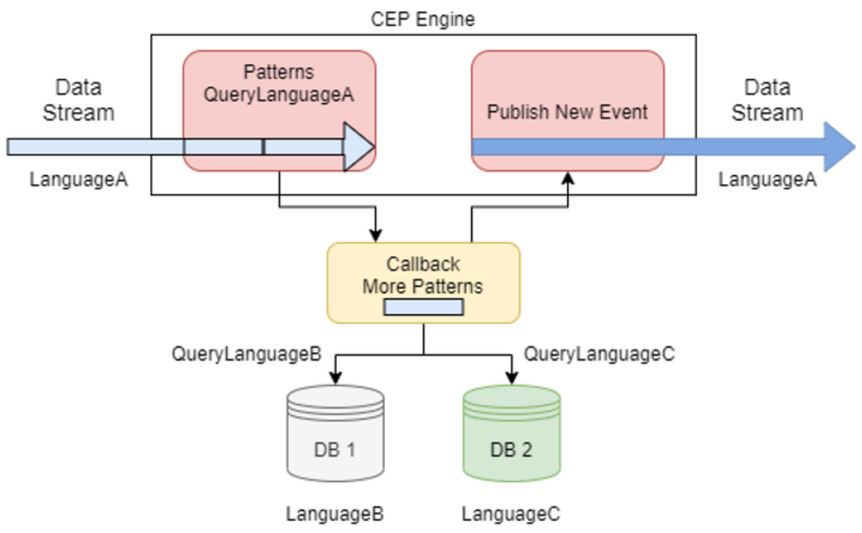

Figure 2.1: Traditional CEP query combining data from the stream with data on external databases.

When using an SCEP engine, the combination of data from the stream with an external KB becomes a simpler task, figure 2.2 illustrates this process. The data on the stream and on the external KB are both represented using RDF, enabling the possibility of writing one single query that can access both the data on the stream and on the external KB at the same time. The drawback is that combining the stream with the KB often causes an increase on processing time due to the insertion of the KB within the stream processing.

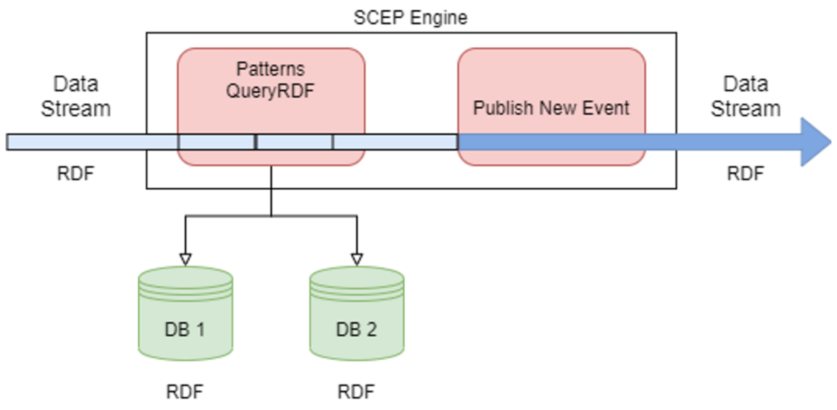

Figure 2.2: SCEP query combining data from the stream with data on an external KB.

Table 2.1 compares some popular RSP engines [34] in light of these four Semantic CEP requirements. The table shows that only SPAseq can be considered a Semantic CEP solution, because it covers all requirements. That said, all listed RSP engines can be used as building blocks for Semantic CEP, and this is precisely the goal of DSCEP, the infrastructure we propose in this thesis. It is important to say that all RSP engines and even SPAseq are standalone engines, they don't provide an infrastructure for distributed processing and to build operator networks. Therefore, in order to connect multiple instances of RSP engines, am distributed infrastructure is still required.

Although all RSP engines listed have problems with scalability in regards to the KB size, or to the complexity of the query, or even because of the high throughput of incoming events, parallelization can be used to reduce the processing time of a query. Additionally, an infrastructure can provide features to approximate current RSP solutions to the processing model of SCEP engines. For example, by offering window management capabilities, support to multiple streams, and support for streams of RDF-graphs.

Table 2.1: Semantic CEP requirements vs RSP engines.

	CQELS[19]	C-SPARQL[16]	EP-SPARQL[18]	$\text{SPARQL}_{\text{Stream}}$[17]	SPA_{seq}[21]
RDF stream: triples/graphs	$+/-$	$+/-$	$+/-$	$+/-$	$+/+$
External KB access	$+$	$+$	$+$	$+$	$+$
Multiple streams (aggregation)	$+$	$+$	$-$	$+$	$+$
Windows: count-based / time-based	$+/+$	$+/+$	$-/-$	$-/+$	$+/+$
Compositional IO	$-$	$-$	$-$	$-$	$+$

3
Conceptual Architecture

We now present the conceptual architecture of our decentralized infrastructure for Semantic CEP. The infrastructure enables the distributed processing of RDF streams using a network of operators.

3.1
Assumptions

These assumptions were created to simplify the discussion and help us to focus more on the main problem of the thesis. They are also reasonable because each one of them address different layers of the system.

In the description of the infrastructure, we assume that:

(i) the events sent through the infrastructure always reach their destination;

(ii) the machines and the software that runs the infrastructure do not fail;

(iii) timestamps increase monotonically (an operator cannot receive an event with a timestamp older than the last event received).

3.2
Infrastructure Modules

The infrastructure is built up from three kinds of modules (see Figure 3.1): Stream Generator, Operator, and Client. Multiple instances of the modules can occur in an instantiation of the infrastructure. The instances execute independently and asynchronously to each other; they can run on different machines and communicate through the publish-subscribe paradigm [36].

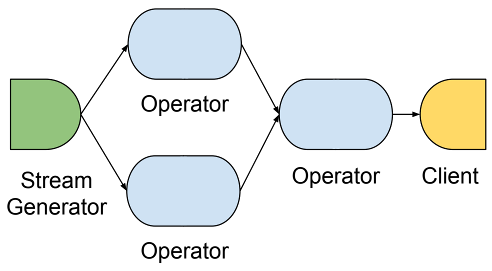

Figure 3.1: An instance of the proposed infrastructure.

The internal structure of each of the three kinds of modules is described in the following subsections.

3.2.1 Stream Generator

The *Stream Generator*, Figure 3.2, generates an RDF stream. It has a Script component that contains the generation logic and a Publisher component that publishes the events generated by the Script. The Publisher can be configured to operate in triple-mode or graph-mode. In triple-mode, each triple fed by the Script delimits a new event in the stream; in graph-mode, events are represented by graphs (sets of triples). When in graph mode, the Publisher ensures that all triples in the same graph have the same timestamp. The KB in figure 3.2 is not part of the infrastructure; it illustrates an external database that the Script component can use in order to generate a data stream.

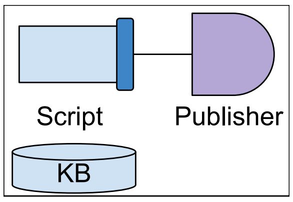

Figure 3.2: Stream Generator.

3.2.2 Operator

The *Operator*, Figure 3.3, is the processing element of the network. It takes as input RDF streams, processes them, and generates resulting output streams. An Operator consists of an Aggregator component, a KB component, and n pairs of RSP engine and Publisher components. The Aggregator aggregates the incoming streams: it takes RDF streams from other Operators or Stream Generators and merges them into a single aggregated stream. The Aggregator continually slices the aggregated stream into windows and feeds these windows to the attached RSP engines. It ensures that the events in the aggregated stream are sorted by timestamp. These can be instances of the same RSP engine or different RSP engines.

The Aggregator can send windows in parallel to the attached RSP engines, enabling a type of processing called Data-Parallel CEP [25], which can boost the overall performance and scalability of the infrastructure. Note that all attached RSP engines must be configured to receive the same window size and type to enable the Aggregator to divide the windows it creates among the RSP

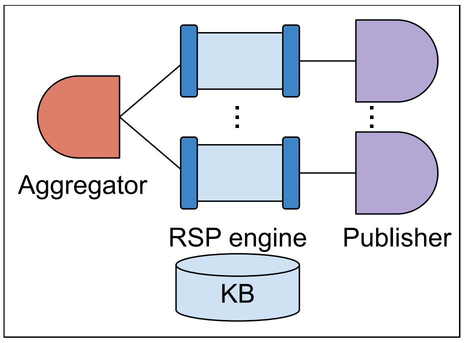

Figure 3.3: Operator module.

engines. The user configures the precise way that the Aggregator will make the window division. Hence, since the Aggregator handles stream aggregation and window partitioning on behalf of operators, RSP engines that do not support specific aggregation/partitioning strategies can be seamlessly integrated into the infrastructure.

In Figure 3.3, each RSP engine processes the windows it gets from the Aggregator (possibly using data from external KBs) and produces related events which its associated Publisher sends. It is the same kind of Publisher component used in the Stream Generator module, Figure 3.2. As before, the Publisher ensures that each outgoing RDF triple has a timestamp and can operate in triple-mode or graph mode. It is worth mentioning that in this infrastructure, the streams generated by a Publisher are always received by an Aggregator.

It is also important to say that our implementation of the Data-parallel CEP paradigm [25] does not include the sequencer. The sequencer is a component of the Data-parallel CEP responsible for coordinating the events detected by the publishers. Since the events are detected in parallel, some inconsistency may appear on the correct sequence of the detected events. For example, let us say windows A and B will be processed in parallel, and window B has a timestamp older than window A. Also, event A will be detected on window A, and event B will be detected on window B. In theory, since window A occurs before window B, event A should also occur before event B. However, in this case, event A may be generated after event B since there is no synchronization (in terms of time) among all RSP engines instances. Thus event A will have a timestamp older than event B's timestamp. This event sequence problem should be evaluated depending on the use case. There are use cases in which this sequence problem would not be an issue, but there are use cases that the ordering of the detected event is essential.

3.2.3 Client

The last module of the infrastructure is the *Client*, Figure 3.4. The *Client* delivers events to end-users. It consists of an Aggregator component (of the same kind used in the Operator) and one or more Scripts. As before, the Aggregator merges multiple input RDF streams into an aggregated stream; this stream is then sliced into windows which are finally passed to the Scripts containing the end-user logic.

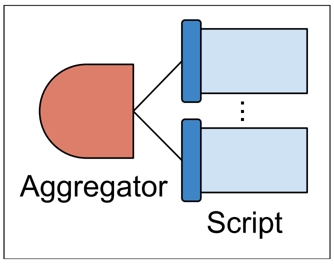

Figure 3.4: Client module.

3.3 Supported query parallelisms

The proposed infrastructure enables two kinds of query parallelism: inter-query and intra-query. It enables inter-query parallelism because different queries can be executed in parallel by different operators, each running its set of RSP engines. Figure 3.5 illustrates the inter-query parallelism, where Query1 is split into three parts that can be parallelized.

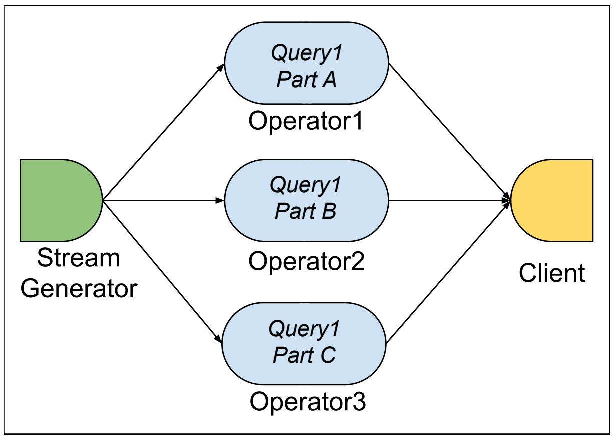

Figure 3.5: Inter-query parallelism.

Note that for the inter-query parallelism to be possible, the query must be split into subqueries (parts) which can be executed in parallel. To divide a query into multiple subqueires is not a simple task and requieres a more complex discussion, in chapter 6 we discuss its limitations and difficulties.

Listing 3.1 is an example of query that can be dividable into three parallel parts and one additional query responsible to join the results. Instead of retrieving all three properties of an instance of MusicalArtist and filtering them using only one query, it is possible to separate the query into three parts in which each of them will retrieve one property and apply a filter.

Part A of the query (listing 3.2) retrieve the genre of the instances of MusicalArtist class if the genre is either Alternative_rock, Pop_music or Rock_music. Part B (listing 3.3) retrieve the names of the instances of MusicalArtist that starts with the letter 'A', and part B retrieves all abstracts of the instances of MusicalArtist that are written in english.

Listing 3.1: Divideble SPARQL query example.

```
PREFIX dbo: <http://dbpedia.org/ontology/>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX dbr: <http://dbpedia.org/resource/>
select ?concept ?genre ?name ?abstract
where
{
?concept rdf:type  dbo:MusicalArtist .
?concept dbo:genre ?genre .
?concept foaf:name ?name .
?concept dbo:abstract ?abstract
FILTER (?genre IN (dbr:Alternative_rock, dbr:Pop_music, dbr:
   Rock_music) )
FILTER( REGEX( ?name , "^A" ) )
FILTER( lang( ?abstract ) = 'en' )
}
```

Listing 3.2: Divideble SPARQL query example: Part A.

```
PREFIX dbo: <http://dbpedia.org/ontology/>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
select ?concept ?genre
where
{
?concept rdf:type  dbo:MusicalArtist .
?concept dbo:genre ?genre .
FILTER (?genre IN (dbr:Alternative_rock, dbr:Pop_music, dbr:
   Rock_music) )
}
```

Listing 3.3: Divideble SPARQL query example: Part B.

```
PREFIX dbo: <http://dbpedia.org/ontology/>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
select ?concept ?name
where
{
?concept rdf:type  dbo:MusicalArtist .
?concept foaf:name ?name .
FILTER( REGEX( ?name , "^A" ) )
}
```

Listing 3.4: Divideble SPARQL query example: Part C.

```
PREFIX dbo: <http://dbpedia.org/ontology/>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
select ?concept ?abstract
where
{
?concept rdf:type  dbo:MusicalArtist .
?concept dbo:abstract ?abstract .
FILTER( lang( ?abstract ) = 'en' )
}
```

The proposed infrastructure also enables intra-query parallelism (Figure 3.6), which can be done by instantiating multiple RSP engines running the same query into one Operator. All RSP engines are connected to one Aggregator, which splits the windows of the same RDF stream among the RSP engines. To split the windows among different RSP engines is a natural way to parallelize the processing of a query in the context of RDF streams. Since RDF stream processing requires splitting the stream into windows, intra-query parallelism can be applied to all queries. Every RSP engine instantiated into the Operator will receive different windows of the same stream.

Finally, when the subqueries execute in different operators, we get what is called *inter-operator parallelism*: every subquery gets the same data stream but executes in different operators. When subqueries are executed within an operator by multiple RSP instances, we get what is called *intra-operator parallelism*: every subquery receives different windows of the same stream.

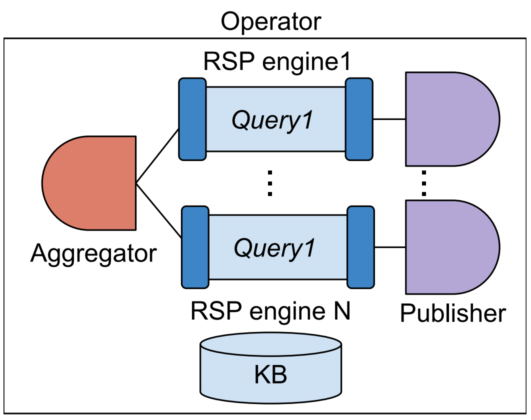

Figure 3.6: Intra-query parallelism.

4
Implementation

In the implementation of DSCEP, we use Apache Kafka[1] for module communication.

4.1
Apache Kafka and DSCEP

Kafka is a fault-tolerant distributed event streaming platform that enables users to publish and subscribe to data streams. DSCEP is written in Java and uses two different Kafka APIs: Producer API and Consumer API. The Producer API allows the user application to publish a data stream to one or more Kafka topics. In contrast, the Consumer API allows the user application to subscribe to a Kafka topic to receive a data stream.

Kafka Topics are a logical category of messages; they represent a stream of the same data type. In order to receive a specific type of stream, a data stream Kafka consumer must subscribe to the topic in which the data stream is being published. Topics can also be divided into partitions to enhance scalability. Partition is analogous to shard in databases and is the core concept behind Kafka's scaling capabilities. Partitions can divide a data stream of a topic into multiple different and smaller data streams, equal to the number of partitions created. By using partitions, it is possible to have multiple Kafka consumers reading from the same Kafka topic simultaneously. Each consumer will read data from a different partition of the topic. This aspect enchants scalability because it is possible to divide the same data stream into smaller streams and have each of these smaller data streams consumed by a different Kafka consumer.

The Producer is the Kafka client that publishers messages to a topic. Moreover, one of the Producer's responsibilities is to decide which partition to send the messages to. DSCEP used the default configuration provided by Apache Kafka. DSCEP allows Kafka to decide which partition to send the data. Kafka has a logic that tries to balance the total number of messages on all partitions.

[1]Apache Kafka: `https://kafka.apache.org/`

The last central aspect of Kafka is the consumer. A consumer reads messages from the topic's partitions in the same order in which the messages arrive. The topic also works as a data store that can persist data for a pre-configured period. Thus, if a consumer needs to go back and read older messages at any point in time, it can do so by resetting the offset position. DSCEP uses the default period for persisting messages on topics.

Consumers can also work together by using the concept of consumer groups. A consumer group is a set of consumers that read messages from a topic. If two different topics are from different consumer groups, they will receive all messages from the topic. If two consumers must receive all messages from the data stream, they must be from different consumer groups. On the other hand, if two consumers must divide the load of processing the data stream of a topic, they must be from the same consumer group. By setting two consumers to the same consumer group, they will receive different messages from the same data stream on the topic, enabling the processing of messages of the same data stream in parallel.

Section 4.2 explains in detail how each component of DSCEP was implemented and how Kafka is used to enable them to interact with each other. Finally, section 4.3 contains two different examples of how to configure the DSCEP infrastructure for two different operator topologies.

4.2 DSCEP Components

The implementation of each DSCEP module (Stream Generator, Operator, and Client) can use up to four different components: Publisher, Aggregator, RSP engine, and Script.

4.2.1 Publisher Component

The Publisher component of DSCEP's Stream Generator and Operator modules, Figures 3.2 and 3.3, uses Kafka's Producer API. More specifically, each Publisher is a Kafka producer, which the user must create to publish a data stream to one or more Kafka topics. The user specifies these Kafka topics through the DSCEP's configuration file, and they identify to which SCEP operator the message will be delivered.

In the Stream Generator module, the *Publisher* is responsible for receiving the stream produced by the *Script* component and for publishing it on a Kafka topic. The user's *Script* and *Publisher* can be created using any programming language that supports Kafka.

Additionally, the user has to create a name for the Kafka topic and use one of the message formats supported by DSCEP.

Currently, DSCEP supports two message formats: RDF-triple and RDF-graph. The RDF-triple format is used for RDF triple streams, and the RDF-graph format is used for RDF graph streams. Messages are written in JSON.

One advantage of implementing the *Publisher* component using Java is that it is possible to extend the DSCEP's data model representation, written with Java classes, to facilitate the creation of a class that can generate the JSON messages.

Here is an example message in the RDF-triple format:

Listing 4.1: An example message in the RDF-triple format.

```
{"ID": "Window1",
"Producer":    "StreamGenerator_1",
 "isRDFgraph":  "false",
 "Triples": [{
   "Subject":   "http://ex.org/Bob",
   "Predicate": "http://ex.org/tweetAbout",
   "Object":    "http://dbpedia.org/resource/Demi_Lovato",
   "Timestamp": 1000}]}
```

The "ID" attribute provides a unique identification for the message among all the windows of the same RDF stream. For each window that a "Producer" sends, the producer will generate a unique window "ID" that distinguishes the window from all the previous windows sent by the producer of that RDF stream. The "Producer" attribute identifies the message publisher, and the "isRDFGraph" attribute distinguishes between RDF-triple and RDF-graph messages. The "Triples" attribute carries the payload. If the message is in RDF-triple format (i.e., if "isRDFGraph" is false), its payload is a list of timestamped RDF triples. Otherwise, the payload is a list of RDF graphs, where each graph is a list of timestamped triples. The attributes "ID" and "Producer" are automatically generated by the DSCEP's *Publisher* component.

Messages in the RDF-graph format have the same initial (header) attributes, but the "Triples" attribute contains a list of RDF-graphs (sets of triples), and the "isRDFGraph" attribute is set to true. Here is an example:

Listing 4.2: An example message in the RDF-graph format.

```
{"ID": "Window1",
 "Producer":    "StreamGenerator_2",
 "isRDFgraph":  "true",
 "Triples": [
[{
   "Subject":   "_node0",
   "Predicate": "rdf:type",
   "Object":    "sioc:post",
   "Timestamp": 1000
},
{
   "Subject":   "_node0",
   "Predicate": "schema:mentions",
   "Object":    "_node1",
   "Timestamp": 1000
},
{
   "Subject":   "_node1",
   "Predicate": "nee:hasMatchedURI",
   "Object":    "dbp:Greyson_Chance",
   "Timestamp": 1000
}],
[{
   "Subject":   "_node2",
   "Predicate": "rdf:type",
   "Object":    "sioc:post",
   "Timestamp": 1050
},
{
   "Subject":   "_node2",
   "Predicate": "schema:mentions",
   "Object":    "_node3",
   "Timestamp": 1050
},
{
   "Subject":   "_node3",
   "Predicate": "nee:hasMatchedURI",
   "Object":    "dbp:Ed_Sheeran",
   "Timestamp": 1050
}]
]}
```

This message has two RDF-graphs represented on it; each RDF-graph has three triples and represents a single tweet with its timestamp. Each triple of an RDF graph conveys only part of the information in the tweet and has the same timestamp. The RDF graph containing triples with the timestamp equal to 1000ms represents information about a tweet that mentions the entity dbp:Greyson_Chance; the other RDF graph with a timestamp equal to 1050ms represents information about a tweet that mentions the entity dbp:Ed_Sheeran.

4.2.2 Aggregator, RSP engine and Script Components

The Aggregator, RSP engine, and Script components of DSCEP's Operator and Client modules, Figures 3.3 and 3.4, use Kafka's Consumer API. All RSP engines of the same Operator are in the same Kafka consumer group. So, each event received by an operator is processed exactly once by one of its RSP engines. Kafka's parallel processing semantics guarantees that a consumer does not get an event that another consumer in the same group has already processed. Similarly, all Scripts of the same Client are in the same consumer group.

Figure 4.1 illustrates how the Aggregator and Publisher modules are connected. An Aggregator subscribes to one or more Publisher topics. Within this infrastructure, the only component that can read data produced by a Publisher is the Aggregator. It consumes messages from these Publisher' topics, aggregates them, and splits the aggregated message stream into windows. The Aggregator has a mapping table that tells which RSP engine each received RDF stream should be sent. The windows produced by an Aggregator are published to the corresponding topic to which the RSP engines subscribe.

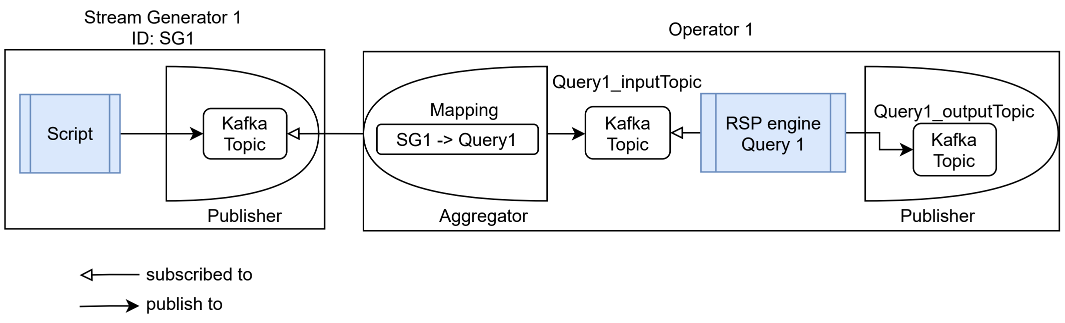

Figure 4.1: Aggregator and Publisher communication.

All groups of windows published by an Aggregator have the same JSON format as those published by the Publisher component. The Aggregator just ensures that the number of events on the window is according to the window size configured by the user. The subscription of Aggregators to Publisher topics, RSP engines to Aggregator topics, window size, etc., are done via a configuration file. The subscription is dynamic and allows us to attach new modules (Stream Generators, Operators, and Clients) to the system on the fly. The only drawback is that newly attached modules do not receive past events, as DSCEP does not persist past events.

Here is an example of the configuration file.

Listing 4.3: An example of the configuration file.

```
1 id = 1
2 queryIDs = 1, 2, 14
3 query1_inputStreamTopics = IS1
4 query1_inputTopic = Q1I
5 query1_outputTopic = Q1O
6 query1_windowInfo = [RANGE TRIPLES 2000]
7
8 query2_inputStreamTopics = IS1
9 query2_inputTopic = Q2I
10 query2_outputTopic = Q2O
11 query2_windowInfo = [RANGE TRIPLES 2000]
12
13 query14_inputStreamTopics = IS1
14 query14_inputTopic = Q14I
15 query14_outputTopic = Q14O
16 query14_windowInfo = [RANGE TRIPLES 2000]
```

The configuration file is a .properties file; which is recognized by the class java.util.Properties, entries in this file are written in a single line. The `id` attribute (line 1) provides a unique identification of the operator where the aggregator is running. The `queryIDs` attribute (line 2) is a list of query identifiers representing the queries that this operator will execute. For each query executed by the operator, there will be four different attributes, each of them with its respective *ID* value.

- `query{ID}_inputStreamTopics`: Contains a list of Kafka topics in which the query with the number ID must be subscribed. This attribute define all input streams for this query.
- `query{ID}_inputTopic`: The kafka topic name that the query with number ID is subscribed to.
- `query{ID}_outputTopic`: The kafka topic name that the query with number ID will publish the output stream.
- `query{ID}_windowInfo`: This attribute details to the aggregator how to create the windows for the query with number ID. `[RANGE TRIPLES 2000]` means that it the query will receive windows of 2000 triples each.

Finally, an RSP engine of an Operator can connect to one or more external KBs. These KBs are considered external resources and are not managed by DSCEP—currently, the system has no built-in database manager and cannot split KBs automatically. A built-in database in DSCEP would be an addition because RSP engines could have the option to use it in order to manipulate the RDF triples from the stream windows and process them. To split KBs automatically based on a query is sure a helpful feature to offer, but it is yet an open research problem; we discuss some challenges and limitations of it in chapter 6.

4.2.2.1 Aggregator's window management

In the aggregator's current implementation, the only available type of window is the count window. The RDF stream is divided by the Aggregator into a specific number of triples defined by the user. In the window creation process, if the RDF stream is in the RDF-graph format, the aggregator ensures that every RDF-graph fits on the window size. For example, if the window size is 2000 RDF triples and there is not enough space to add another RDF-graph to this window, the Aggregator will leave the next RDF-graph to the next window.

The RSP engine component of DSCEP was built to accept different implementations of RDF stream processors. Some of them work with RDF triple streams and only process the window when the number of RDF triples that arrived to be processed is equal to the window size. Thus, building windows with a size less than its configured size to be processed by some specific RSP engine was a challenge.

To ensure that the Aggregator can work with any RDF stream processor and process windows with fewer triples than the expected/configured size, the Aggregator adds dummy triples to the window to match the total size expected by the RSP engine and guarantees that the RSP engine will not split the RDF-graphs. The dummy triple is necessary even for the RSP engines that process windows with less than the window size, but wait some time period until they start to process the window content. That is because the next RDF triples that will arrive will belong to another graph, and the following graph will not fit on the current window. These dummy triples added by the Aggregator to complete the window size are meaningless. They all have the same subject, predicate, and object. The dummy triple is (ex:sss, ex:ppp, ex:ooo).

Note that there is a problem with estimating the window size for RDF graph streams. If the user configures a window size smaller than the size of at least one RDF graph of the stream, the aggregator will not be able to send this RDF graph to be processed because triples of an RDF graph cannot be separated. An optimal solution for handling this would be to estimate the size of the window based on the input RDF graph stream and dynamically adjust the window size processed by the RDF stream processor. However, most RDF stream processors do not have support for adjusting the window size of a query on-demand. The only way to do this is if DSCEP does not accept any third-party RDF stream processors and uses one implemented exclusively for DSCEP that accepts the window size change on demand.

Thus, to work with RDF stream processors that do not change window size on demand, the system would have to estimate the window size before the processing starts. To estimate the window size is a challenge because it requires the pre-processing of all RDF graphs of the stream that theoretically has infinite size.

Since DSCEP was created to allow different RDF stream processors implementations, we could not automatically estimate the window size. For these cases of RDF graph streams, to ensure that all RDF graphs will not be divided, the Aggregator discards all RDF graphs that do not fit on the window and tells the user via the log file that a specific RDF graph got discarded.

It is important to say that sometimes using count windows can be difficult depending on the problem, and using other types of windows (e.g., time-window) could be more helpful. That is one reason for the existence of different types of windows, to enable modeling different types of problems. For example, estimating the window size on RDF graph streams could be avoided if the user can model it with time windows. Since all triples inside an RDF graph have the same timestamp, it would force all triples of the same RDF graph not to get split, possibly, making the time window more suitable for adapting an RDF stream processor that only works with triple streams to also work with RDF graphs streams.

Currently, the aggregator is not implemented to work with time windows. The implementation of time windows was not required to evaluate and study our research questions; therefore, we chose not to implement it to focus on the main aspects of the research.

4.3 Configuring an example operator topology on DSCEP

To give a more concrete example of how to configure an operator topology on DSCEP, we give an example of two different operator topologies in this section. The first topology is presented in Figure 4.2 and consists of three operators, one stream generator, and one client module to consume the result stream. Additionally, the following are the Kafka topic names for each module and query:

– Stream Generator: Writes its output stream on the Kafka topic named IS1.

– Query 1: Receives its input stream on the Kafka topic Q1I and produces its output stream on the Kafka topic Q1O.
– Query 2: Receives its input stream on the Kafka topic Q2I and produces its output stream on the Kafka topic Q2O.
– Query 3: Receives its input stream on the Kafka topic Q3I and produces its output stream on the Kafka topic Q3O.
– Query 4: Receives its input stream on the Kafka topic Q4I and produces its output stream on the Kafka topic Q4O.

Note that these Kafka topic names are user-created since the user can create stream generators, RSP engines to process the RDF streams, and clients to consume RDF streams.

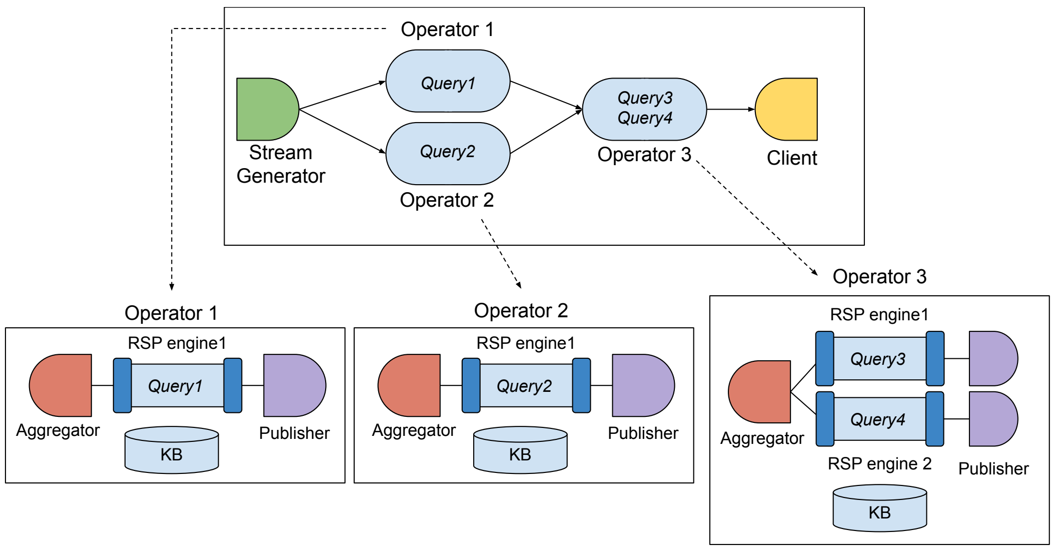

Figure 4.2: First example operator topology.

Operator 1 has one query, which is Query 1, that receives its input stream from the Stream Generator. Query 1 works with a count window, and each window has 2000 triples. Below is the configuration file for the Aggregator of Operator 1:

Listing 4.4: Configuration file for the Aggregator of Operator 1 - First topology.

```
id = 1
queryIDs = 1
query1_inputStreamTopics = IS1
query1_inputTopic = Q1I
query1_outputTopic = Q1O
query1_windowInfo = [RANGE TRIPLES 2000]
```

Operator 2 is similar to Operator 1 and has one query, named Query 2, which receives its input stream from the Stream Generator. Query 1 works with a count window, and each window has 1500 triples. Here is the configuration file for the aggregator of Operator 2:

Listing 4.5: Configuration file for the Aggregator of Operator 2 - First topology.

```
id = 2
queryIDs = 2
query2_inputStreamTopics = IS1
query2_inputTopic = Q2I
query2_outputTopic = Q2O
query2_windowInfo = [RANGE TRIPLES 1500]
```

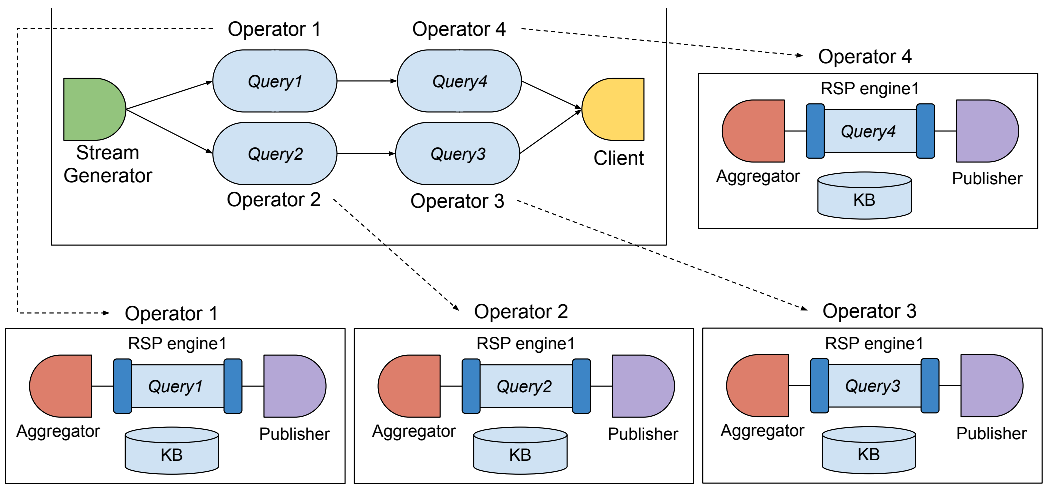

Figure 4.3: Second example operator topology.

Finally, Operator 3 has two different queries (Query 3 and Query 4) and receives two different input streams, Query 1 produces one, and Query 2 produces the other. Query 3 works with a count-window, and each window has 1000 triples, and Query 4 works with a count-window, and each has 500 triples. Here is the configuration file for the aggregator of Operator 3:

Listing 4.6: Configuration file for the Aggregator of Operator 3 - First topology.

```
id = 3
queryIDs = 3, 4
query3_inputStreamTopics = Q1O, Q2O
query3_inputTopic = Q3I
query3_outputTopic = Q3O
query3_windowInfo = [RANGE TRIPLES 1000]

query4_inputStreamTopics = Q1O, Q2O
query4_inputTopic = Q4I
query4_outputTopic = Q4O
query4_windowInfo = [RANGE TRIPLES 500]
```

The second operator topology is presented in Figure 4.3 and consists of one stream generator, one client and four operators, one for each different query.

The difference between the second topology when compared to the first topology is that now query 3 and query 4 are not on the same operator; they are on Operator 3 and Operator 4, respectively. Consequently, there will now be four configuration files, as each operator needs one. The configuration file for Operator 1 and Operator 2 remains the same because they have the same queries and the same window configuration.

Here is the configuration file for Operator 3 for this second topology:

Listing 4.7: Configuration file for the Aggregator of Operator 3 - Second topology.

```
id = 3
queryIDs = 3
query3_inputStreamTopics = Q2O
query3_inputTopic = Q3I
query3_outputTopic = Q3O
query3_windowInfo = [RANGE TRIPLES 1000]
```

The difference is that query 4 is now located on Operator 4, and because of that, it does not appear on Operator 3's configuration file. All information of query 3 from the first example topology stays the same. Finally, this is the configuration file of Operator 4 of the second topology:

Listing 4.8: Configuration file for the Aggregator of Operator 4 - Second topology.

```
id = 4
queryIDs = 4
query4_inputStreamTopics = Q1O
query4_inputTopic = Q4I
query4_outputTopic = Q4O
query4_windowInfo = [RANGE TRIPLES 500]
```

Note that the second topology has only one query per operator and one Aggregator per query.

Besides configuring the Aggregator, the user:

1. writes the stream generation logic and publishes it using the DSCEP's JSON format for RDF-triples or RDF-graphs.

2. writes the Client module's Script component with the logic for consuming the final RDF stream.

3. writes the RSP engine logic, which contains the queries, to process each input RDF stream.

There are some already implemented Operator modules for DSCEP which the user can choose to use. In the next section, each of these Operators is presented.

4.4 DSCEP Implemented Operators

There are two different implemented operators in DSCEP; both are written in Java. One of them is an Operator which uses the C-SPARQL RDF stream processor [16] version 0.9.7[2], which we call by *RSP:CSPARQL Operator*. The other implemented operator uses an RDF stream processor implementation produced in the course of this research, which we call *RSP:BasicProcessor Operator*.

Both RDF stream processor logics, for *RSP:CSPARQL Operator* and *RSP:BasicProcessor Operator* were written in the RSP engine component of the Operator's module.

The *RSP:BasicProcessor Operator* uses the Apache Jena library version 4.1.0 to query and handle RDF data. Figure 4.4 illustrates how the *RSP:BasicProcessor Operator* is implemented.

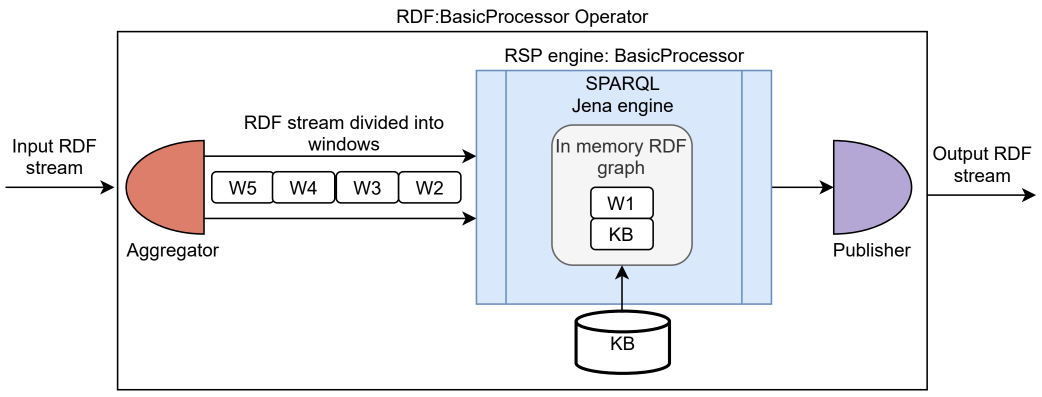

Figure 4.4: *RSP:BasicProcessor Operator* implementation.

First, as required to all operators, the Aggregator is configured for window management and splits the RDF data stream into windows with pre-determined size by the user. For every window that arrives in the RSP engine component (represented in blue on figure 4.4), we insert the window triples with all KB triples using the RDF API of Jena[3] into an RDF graph in memory. We query this RDF graph using Jena ARQ[4] SPARQL and retrieve the result of the query. After the result is obtained, the window is deleted from the RDF graph built-in memory so as to prepare the RSP engine for the next window to arrive. This

[2]C-SPARQL: `http://streamreasoning.org/resources/c-sparql`
[3]Jena's RDF API: `https://jena.apache.org/documentation/rdf/index.html`
[4]ARQ (SPARQL): `https://jena.apache.org/documentation/query/index.html`

step is repeated for each window to be processed. All query results are sent to the Publisher component, which publishes those results in the corresponding Kafka topic.

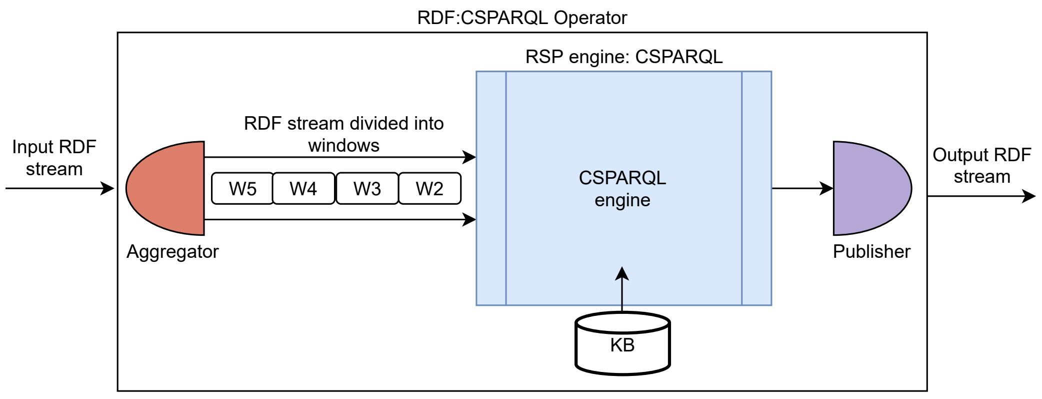

Figure 4.5: *RSP:CSPARQL Operator* implementation.

Figure 4.5 illustrates how the *RSP:CSPARQL Operator* was implemented. The difference between the *RSP:CSPARQL Operator* implementation and the *RSP:BasicProcessor Operator* is the content of the RSP:engine component. For the *RSP:CSPARQL Operator* we wrote the code to process the input stream using CSPARQL[16].

Listing 4.9 shows all the parameters that can be passed to both operators, by using an input file called *operatorInput.properties*:

Listing 4.9: *operatorInput.properties* file.

```
query = "..." # continuous query used by the operator on the input stream
idProperty = "sioc:id"
graphQuery = "..." # query that retrives the RDF-graph with the idProperty
```

Line 1 contains the query that the operator will apply to the input stream. The file *operatorInput.properties* is the same for both implemented operators, *RSP:BasicProcessor Operator* and *RSP:CSPARQL Operator*. The difference is that for the *RSP:CSPARQL Operator*, the `query` attribute must contain a query written in the CSPARQL language, and for the *RSP:BasicProcessor Operator*, the `query` attribute must contain a query written in the SPARQL language.

In order to give one example on how to use the *RSP:CSPARQL Operator*, listing 4.10 shows one CSPARQL example query that the user can give as input:

Listing 4.10: CSPARQL example query.

```
REGISTER STREAM TweetStream AS
PREFIX ...
CONSTRUCT {
?tweet sioc:id ?id .
```

```
 5 ?tweet schema:mentions ?entity .
 6 ?entity nee:hasMatchedURI ?artistURI .
 7 ?artistURI rdf:type dbo:MusicalArtist .
 8 }
 9 FROM STREAM <.../Tweets> [RANGE TRIPLES 2000]
10 FROM <.../KB.rdf>   # (The local Knowledge Base to access.)
11 WHERE {
12 ?tweet rdf:type sioc:Post .
13 ?tweet sioc:id ?id .
14 ?tweet schema:mentions ?entity .
15 ?entity nee:hasMatchedURI ?artistURI .
16 ?artistURI rdf:type dbo:MusicalArtist .
17 }
```

This CSPARQL query will match every tweet post that mentions a dbo:MusicalArtist. In line 9 one can see the definition of the window type (which is count-window) and its size (2000 triples per window). This information must also be in the configuration file of the Operator because the aggregator is the element responsible for window partitioning within DSCEP.

Besides the query, if the stream's events are RDF-graphs, the user must provide two additional parameters. These are the parameters described on line 2 and line 3 of the *operatorInput.properties* file showed above in listing 4.9.

1. Parameter `idProperty`: The RDF property which identifies the RDF-graph. This information is used by a SPARQL query which will retrieve all RDF-graphs identifiers from a given set of triples.

2. Parameter `graphQuery`: A SPARQL query capable of retrieving all triples of a RDF graph. This query uses the `idProperty` to identify each RDF-graph.

For example, let us consider that the user has an RDF-graph stream with events that use the RDFS model described in Figure 4.6.

All RDF-graphs of this stream are identified by the property (`sioc:id`), thus this RDF property must be the value of the parameter `idProperty` of the *operatorInput.properties* file. The `graphQuery` parameter is the SPARQL query capable of retrieving all triples of an RDF-graph given its identifier. Listing 4.11 is an example for the `graphQuery` parameter:

Listing 4.11: Example SPARQL query for the *graphQuery* parameter.

```
REGISTER STREAM TweetStream AS
PREFIX ...
CONSTRUCT {
?tweet sioc:id ?id .
?tweet schema:mentions ?entity .
?entity nee:hasMatchedURI ?artistURI .
?artistURI rdf:type dbo:MusicalArtist .
}
FROM STREAM <.../Tweets> [RANGE TRIPLES 2000]
FROM <.../KB.rdf>   # (Only for local access.)
WHERE {
?tweet rdf:type sioc:Post .
?tweet sioc:id ?id .
?tweet schema:mentions ?entity .
?entity nee:hasMatchedURI ?artistURI .
?artistURI rdf:type dbo:MusicalArtist .  # (triple accessed on the KB.)
}
```

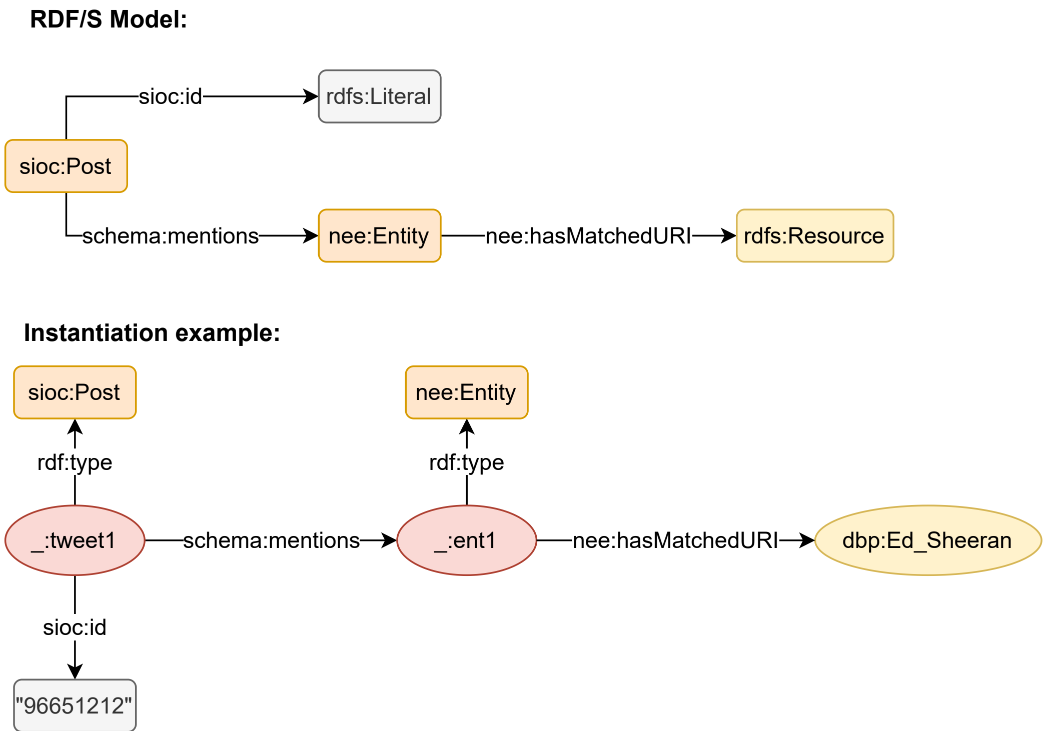

Figure 4.6: RDFS model of an RDF-graph and one example of an instance.

DSCEP with these two parameters (`idProperty` and `graphQuery`), can identify all RDF-graphs contained on a set of triples. Note that this implies all RDF-graphs must have at least one triple that can identify the whole graph.

5 Evaluation

To evaluate DSCEP, we prepared three different experiments. The first experiment is focused on comparing the processing time of a single and complex query (Q) deploying and executing it in two modes: first as a monolithically, second as dividing this same query into six smaller queries (Q1-Q6) and executing it with DSCEP using a network of operators that when combined produce the same results of query Q. The second experiment is focused on measuring the impact of varying the total KB size versus search-space size in subqueries Q1 and Q2. Q1 and Q2 are queries that combine data from the stream with data on a knowledge base. Furthermore, the third experiment is focused on evaluating the intra-operator parallelism and check whether and in which conditions the intra-operator parallelism can decrease query processing time.

5.1 Input RDF Stream

For the evaluation of DSCEP, we used the TweetsKB [37], a RDF dataset containing anonymized and annotated tweets (Tweeter posts). We took a subset corresponding to one month of tweets (February 2013)[1] and transformed it into a stream by grouping the triples of each tweet into an RDF graph. Each graph is timestamped with the date and time of the tweet's creation. The resulting RDF stream consisted of approximately 60,000 tweets (graphs) or 2.3 million triples.

Among other things, each tweet in the resulting stream contains:

(i) an *id* which uniquely identifies it;

(ii) references to *entities* which it mentions;

(iii) a *sentiment analysis* score which is a numeric value in the interval [-5.0,5.0] (from bad to good);

(iv) *likes and shares* counts counting the number of times the tweet received like and was shared.

[1]TweetsKB: urlhttps://data.gesis.org/tweetskb/

5.2 Knowledge Base

As the background knowledge, we used DBpedia, an RDF knowledge base extracted from Wikipedia. The version we used has approximately 368 million triples and was extracted from DBpedia's public SPARQL endpoint in June 2019. We chose DBpedia because TweetsKB uses it to annotate the tweets (the entities referred to in tweets are DBpedia resources).

In the experiments, when we refer to the *total KB size* ($\mathrm{KB}_{\mathrm{total}}$) we mean the total number of triples in the KB. Moreover, when we refer to the *search-space size* ($\mathrm{KB}_{\mathrm{used}}$) we mean the number of triples that actually need to be considered to answer a given query. For instance, to answer the query "select all musical artists born before 1975", the evaluator only needs to consider the 103,075 triples comprising the musical artist subgraph of DBpedia (and not the whole DBpedia).

It is important to take into account that the KB used in the experiment is lightweight, containing a flat data model which makes less complex the process of KB partitioning. One may ask why to use the RDF format for flat data models when it is possible to use the relational model, which is proven to be much faster. The RDF format does not only have the advantage of providing reasoning/inferences, it is also widely used in the literature as a neutral format to integrate different data sources [38, 39].

5.3 Access Methods

We compared two different methods for accessing the KB. In the *local access method* ($\mathrm{KB}_{\mathtt{LOCAL}}$), the KB is passed to C-SPARQL (the RSP engine we used in the experiments) as an RDF file using the `FROM` directive of SPARQL. This causes C-SPARQL to fill up every window with all triples in the KB, which is simply impractical when the KB is the whole DBpedia. Hence, to be able to run the experiments, specifically when using the local access method, we used a reduced KB containing only the musical artist and television show subgraphs of DBpedia. This reduced KB contained 132,489 triples (approximately 0.03% of DBpedia).

The second method we used for accessing the KB is the *endpoint access method* ($\mathrm{KB}_{\mathtt{GLOBAL}}$). In this method, the KB containing the whole DBpedia is kept in a triplestore (RDF database) which is accessed by C-SPARQL using the `SERVICE` directive of SPARQL. Using this method, we can make C-SPARQL access the desired subgraphs of DBpedia on the fly, without adding the whole

KB to every window–although the subgraphs resulting from the `SERVICE` call are still added to every window.

5.4 Setup

For experiments 1 and 2, we deployed DSCEP on a machine with 512 GB of RAM and two AMD EPYC 7451 processors, each with 24 cores, 2.3 GHz, and 64 MB of cache. For the communication infrastructure, we used Apache Kafka version 2.0 and Zookeeper[2]. We used Docker[3] to run the DSCEP modules (one module per Docker container). Moreover, we used the Virtuoso[4] triplestore to run our own DBpedia endpoint in a separate Docker container.

Also, for experiments 1 and 2, as RDF stream processor, we used C-SPARQL [16] version 0.9.7[5] with one C-SPARQL instance per DSCEP operator. We configured C-SPARQL to use count-based windows (with 1000 triples per window) in the experiments and delegated to DSCEP the responsibility of slicing" (cutting) the graphs and creating windows from these slices. With this setup, DSCEP ensures that triples from one single graph do not end up in different C-SPARQL window.

5.5 Experiments

First experiment

We considered a complex C-SPARQL query Q and compared its evaluation using a single operator versus the evaluation of smaller queries Q_1–Q_6 by a network of operators that, when combined, produced the same result as query Q. The goal here was to show that we can reduce the overall processing time of a complex query by splitting it into smaller, simpler queries, using DSCEP to coordinate their execution and combine their results.

The general structure of query Q is shown below, and its complete version can be found in Appendix A.1.

```
REGISTER STREAM TweetStream AS
PREFIX ...
# Get all tweets mentioning both musical artists and TV shows
# plus the aggregated sentiment scores, likes, and shares.
CONSTRUCT { ... }
```

[2]Apache Zookeeper: `https://zookeeper.apache.org/`
[3]Docker: `https://www.docker.com/`
[4]Virtuoso: `https://virtuoso.openlinksw.com/`
[5]C-SPARQL: `http://streamreasoning.org/resources/c-sparql`

```
 6 FROM STREAM <.../Tweets> [RANGE TRIPLES 1000]
 7 FROM <.../KB.rdf>   # (Only for local access.)
 8 WHERE {
 9 # $Q_1$: Get all tweets mentioning a musical artist.
10 ...
11 # $Q_2$: Get all tweets mentioning a TV show.
12 ...
13 # $Q_3$: Aggregate sentiment score of musical artists.
14 {SELECT ?artistURI (count(?posNum) as ?cntPositive)
15  WHERE {...} GROUP BY ?artistURI}
16 {SELECT ?artistURI (count(?negNum) as ?cntNegative)
17  WHERE {...} GROUP BY ?artistURI}
18
19 # $Q_4$: Aggregate likes/shares of musical artists.
20 {SELECT ?artistURI  (count(?likeNum) as ?cntLikes)
21  WHERE {...} GROUP BY ?artistURI}
22 {SELECT ?artistURI (count(?shareNum) as ?cntShares)
23  WHERE {...} GROUP BY ?artistURI}
24
25 # $Q_5$: Aggregate sentiment score of TV shows.
26 {SELECT ?tvshowURI (count(?posNum) as ?cntPositive)
27  WHERE {...} GROUP BY ?tvshowURI}
28 {SELECT ?tvshowURI (count(?negNum) as ?cntNegative)
29  WHERE {...} GROUP BY ?tvshowURI}
30
31 # $Q_6$: Aggregate likes/shares of TV shows.
32 {SELECT ?tvshowURI  (count(?likeNum) as ?cntLikes)
33  WHERE {...} GROUP BY ?tvshowURI}
34 {SELECT ?tvshowURI (count(?shareNum) as ?cntShares)
35  WHERE {...} GROUP BY ?tvshowURI}}
```

Query Q is a construct query (line 5). Its result is a new RDF graph containing all tweets from the input stream that mention either a musical artist (line 10) or a TV show (line 12) and also some aggregated statistics: the sentiment score and like and share counts associated with each musical artist (lines 14–17 and 20–23) and TV show (lines 26–29 and 32–35) mentioned in all such tweets. The query uses the external KB to determine whether a resource mentioned by a tweet is a musical artist or TV show (line 7).

Note that the goal of query Q is to produce an RDF graph that can be used for further analyses correlating tweets that mention musical artists and TV shows. For instance, the result can be used to determine which TV shows are good publicity for a given musical artist or which musical artists might be bad guests for a given type of TV show.

We executed query Q in DSCEP using a single stream generator and a single C-SPARQL operator. The results for this configuration are shown in

Table 5.1 under the label Q. Using the local access method ($KB_{\texttt{LOCAL}}$), the KB contains only the subgraphs for musical artists and TV shows; hence KB_{total} is 132,489 and every one of these triples is considered by Q, and so KB_{used} is also 132,489. Using the endpoint access method ($KB_{\texttt{GLOBAL}}$), the KB contains the whole DBpedia (hence KB_{total} is 368,720,213 triples) but KB_{used} is still 132,489 because only the musical artist and TV show subgraphs are considered. The processing time per window was on average 117s for $KB_{\texttt{LOCAL}}$ and 104.3s for $KB_{\texttt{GLOBAL}}$.

Table 5.1: Processing time of Q vs Q_1–Q_6.

	Q		Q_1–Q_6	
	$KB_{\texttt{LOCAL}}$	$KB_{\texttt{GLOBAL}}$	$KB_{\texttt{LOCAL}}$	$KB_{\texttt{GLOBAL}}$
KB_{total}(number of triples)	132,489	368,720,213	132,489	368,720,213
KB_{used}(number of triples)	132,489	132,489	132,489	132,489
Processing time (seconds)*	117	104.3	84.6	81.3

*Seconds per window; average of 10 runs with 1168 windows per run.

To parallelize Q, we divided it into six subqueries Q_1–Q_6, corresponding to the marked regions in the previous listing for Q, and constructed a network of C-SPARQL operators in DSCEP to coordinate their execution. The layout of this network is depicted in Figure 5.1. Each subquery runs in a separate operator with one Docker container per operator. Subqueries Q_1 and Q_2 get all tweets that mention some musical artist and some TV show, respectively. These are the only subqueries that access the KB. Subqueries Q_3 and Q_4 aggregate the sentiment score and like and share counts for the musical artists mentioned in the tweets selected by Q_1, while subqueries Q_5 and Q_6 do the same thing for the tweets mentioning TV shows selected by Q_2. The combined outputs of Q_3–Q_6 are delivered to the client module (in yellow) and comprise the resulting graph, identical to the graph produced by Q. All subqueries run in parallel.

The results for Q_1–Q_6 are shown in Table 5.1. As expected, the total KB size (KB_{total}) and the search-space size (KB_{used}) are the same as those for Q in both access methods. The processing time using local KB access ($KB_{\texttt{LOCAL}}$) was 29% smaller for Q_1–Q_6 (84.6s per window) when compared with Q (117s per window). With endpoint access ($KB_{\texttt{GLOBAL}}$), the reduction was 23% (from 104.3s for Q to 81.3 for Q_1–Q_6).

In both cases, the bulk of the processing time was spent in the subqueries Q_1 and Q_2, which are the only subqueries that access the KB. For instance, the time of 84.6s per window with local KB access is dominated by Q_1 alone—query Q_2 took 26.65s per window, while all of the remaining subqueries took less than

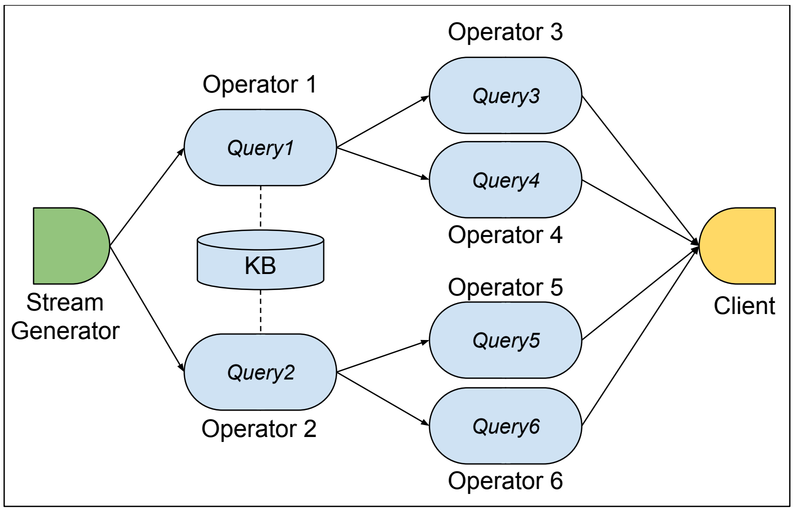

Figure 5.1: The parallelization of Q into six subqueries.

40ms per window. Since all queries run in parallel, it is Q_1 that determines the average processing time per window of the network.

This first experiment demonstrates that dividing a query into parallel subqueries can reduce overall processing time. It also indicates that KB access is costly. The following experiment assesses this cost.

Second experiment

We measured the impact of varying the total KB size (KB_{total}) versus search-space size (KB_{used}) in subqueries Q_1 and Q_2 of the first experiment. The goal here was twofold. First, we wanted to assess the contribution of the search-space size to the overall processing time of the queries. As expected, it takes more time to traverse a more extensive search space. The second and possibly more interesting goal was to highlight that some RSP engines (including C-SPARQL) are highly sensitive to noise (defined as irrelevant or meaningless data [40]) in the KB. In other words, we can degrade their performance by simply growing KB_{total} while keeping KB_{used} fixed. Hence, the ability to control which parts of the KB are accessed by a given query is crucial for obtaining a good performance.

We considered three situations, (a), (b), and (c). The results for each of these are given in Figure 5.2.

In (a), we used the endpoint access method and evaluated queries Q_1 and Q_2 over a KB consisting of the whole DBpedia. So KB_{total} was fixed and equal to approximately 368 million triples. We tweaked Q_1 and Q_2 to determine how variations in the search-space size (KB_{used}) affected the queries. It took Q_1 on average 81.3s to process each window in a search-space containing the whole

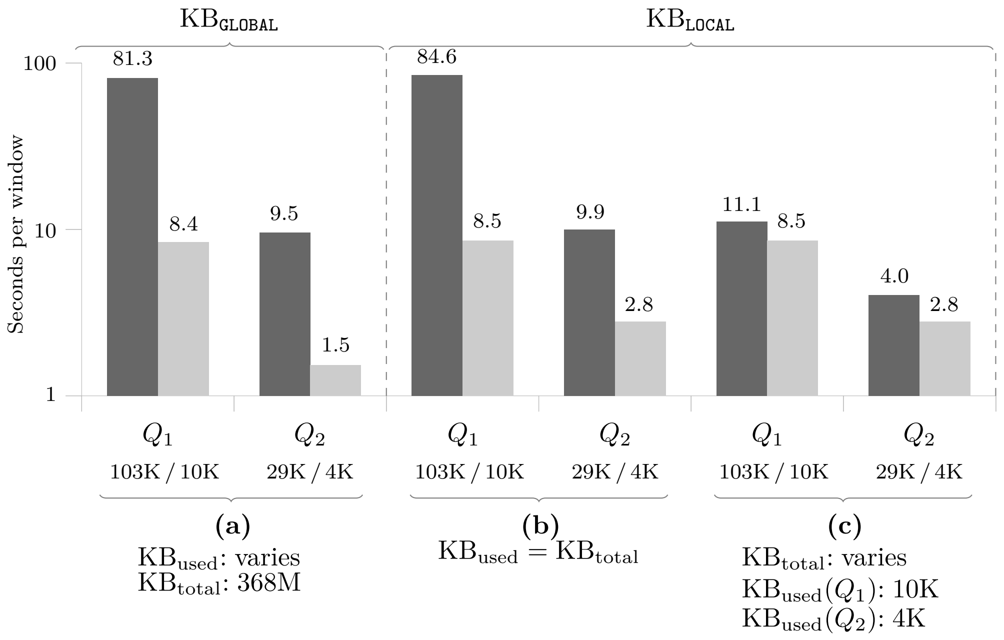

Figure 5.2: Average processing time per window with varying KB_{used} and KB_{total} for Q_1 and Q_2. **(a)** Using the endpoint access method for Q_1 and Q_2 with KB_{total} fixed at 368M triples and KB_{used} with 103K vs 10K triples for Q_1 and 29K vs 4K triples for Q_2. **(b)** Using the local access method for Q_1 and Q_2 with $KB_{used} = KB_{total}$ and 103K vs 10K triples for Q_1 and 29K vs 4K triples for Q_2. **(c)** Using the local access method for Q_1 with KB_{used} fixed at 10K and KB_{total} with 103K vs 10K triples, and for Q_2 with KB_{used} fixed at 4K and KB_{total} with 29K vs 4K triples. The figures for KB_{used} and KB_{total} were divided by 1000 and rounded and the processing times are an average of 4 runs with 1168 windows each; the y-axis is in log scale.

musical artist subgraph of DBpedia ($KB_{used} = 103K$). When we divided this search space roughly by ten ($KB_{used} = 10K$), it took Q_1 ten times less time (8.4s) to process each window on average. Similar behavior was observed for Q_2 as depicted in Figure 5.2(a). These results indicate that restricting the parts of the KB that are accessed by a given query (i.e., reducing its search space) can achieve a significant speedup in processing time.

In (b), we used the local access method with a search-space consisting of the whole KB, i.e., $KB_{used} = KB_{total}$, and we varied both KB_{used} and KB_{total} together. The results of (b) were similar to those of (a). Query Q_1 took 84.6s per window with a search-space and KB size of 103K triples, and 8.5s per window with a search-space and KB with one-tenth of that size (10K triples). Query Q_2 behaved similarly, as shown by Figure 5.2(b). These results, together with those of (a) highlight the direct relation (in the two cases almost linear) between search-space size and processing time.

In (c), we used the local access method, and we kept the search-space size fixed (10K for Q_1 and 4K for Q_2) while considering increasing KB sizes.

Query Q_1 took 11.1s per window on average with a KB containing 103K triples and 8.5s per window on average with a KB containing 10K triples. In this case, by simply adding 90K unrelated triples to the KB, we increased the processing time by 30% (from 8.5s to 11.1s). As shown in Figure 5.2(c), the result was similar for Q_2: the addition of 25K unrelated triples to the KB increased the processing time by approximately 42.8% (from 2.8s to 4s). These increments were not negligible and indicated that the mere presence of unrelated triples in the KB could significantly slow down the query. This is one more reason for partitioning the KB among the operators of a Semantic CEP network.

Third experiment

We deployed DSCEP on a machine with 16GB of RAM, a Quad-Core Intel I7 CPU with 2.7GHz, and four cores for this experiment. Also, for the communication infrastructure, we used Apache Kafka version 2.0 and Zookeeper[6]. As DSCEP operator, we used the *RSP:BasicProcessor Operator*, which uses our implementation of an RDF stream processor detailed in section 4.4.

The goal of this experiment is to evaluate intra-operator parallelism and check whether and in which conditions the intra-operator parallelism can decrease query processing time. We compared the processing time of query Q_7 running it with and without intra-operator parallelism. The goal was to evaluate if we can reduce a query's overall processing time by instantiating multiple RSP engines with the same query and splitting the windows among the RSP engines, using DSCEP for window management, coordinating their execution, and combine their results.

Query Q_7 is shown below:

```
PREFIX ...
CONSTRUCT {?tweet rdf:type sioc:Post .
?tweet sioc:id ?id .
?tweet dc:created ?datetime .
?tweet sioc:has_creator ?postCreator .
?tweet onyx:hasEmotionSet ?emotionSet .
?tweet schema:interactionStatistic ?interactionSet .
?tweet schema:interactionStatistic ?interactionSet2 .
?interactionSet rdf:type schema:InteractionCounter .
?interactionSet schema:interactionType schema:LikeAction .
?interactionSet schema:userInteractionCount ?likeCount .
?interactionSet2 rdf:type schema:InteractionCounter .
?interactionSet2 schema:interactionType schema:ShareAction .
```

[6]Apache Zookeeper: https://zookeeper.apache.org/

```
?interactionSet2 schema:userInteractionCount ?shareCount .
?emotionSet rdf:type onyx:EmotionSet .
?emotionSet onyx:hasEmotion ?positive .
?positive onyx:hasEmotionCategory wna:positive-emotion .
?positive onyx:hasEmotionIntensity ?posNum .
?artistURI ex:hasPositiveNumber ?posNum .
?emotionSet onyx:hasEmotion ?negative .
?negative onyx:hasEmotionCategory wna:negative-emotion .
?negative onyx:hasEmotionIntensity ?negNum .
?tweet schema:mentions ?entity .
?entity nee:hasMatchedURI ?artistURI .
?artistURI dbo:genre ?genre .
?artistURI rdf:type dbo:MusicalArtist .
?entity ex:hasName ?name .
?tweet schema:mentions ?TagClass .
?TagClass rdf:type sioc_t:Tag .
?TagClass rdfs:label ?tag .
?tweet schema:mentions ?UserAcc .
?UserAcc rdf:type sioc:UserAccount .
?UserAcc sioc:name ?userName .}
FROM <.../KB.rdf>   # (The local Knowledge Base to access.)
WHERE
{
?artistURI rdf:type dbo:MusicalArtist .
?artistURI dbo:genre ?genre .
?tweet rdf:type sioc:Post .
?tweet sioc:id ?id .
?tweet dc:created ?datetime .
    ?tweet sioc:has_creator ?postCreator .
    ?tweet onyx:hasEmotionSet ?emotionSet .
    ?tweet schema:interactionStatistic ?interactionSet .
    ?tweet schema:interactionStatistic ?interactionSet2 .
    ?interactionSet rdf:type schema:InteractionCounter .
    ?interactionSet schema:interactionType schema:LikeAction .
    ?interactionSet schema:userInteractionCount ?likeCount .
    ?interactionSet2 rdf:type schema:InteractionCounter .
    ?interactionSet2 schema:interactionType schema:ShareAction
   .
    ?interactionSet2 schema:userInteractionCount ?shareCount .
    ?emotionSet onyx:hasEmotion ?positive .
    ?positive onyx:hasEmotionCategory wna:positive-emotion .
    ?positive onyx:hasEmotionIntensity ?posNum .
    ?emotionSet onyx:hasEmotion ?negative .
    ?negative onyx:hasEmotionCategory wna:negative-emotion .
    ?negative onyx:hasEmotionIntensity ?negNum .
    ?tweet schema:mentions ?entity .
    ?entity nee:hasMatchedURI ?artistURI .
    ?entity nee:detectedAs ?name .
    OPTIONAL { ?tweet schema:mentions ?TagClass }
    OPTIONAL { ?TagClass rdf:type sioc_t:Tag }
    OPTIONAL { ?TagClass rdfs:label  ?tag }
    OPTIONAL { ?tweet schema:mentions ?UserAcc }
    OPTIONAL { ?UserAcc rdf:type sioc:UserAccount }
    OPTIONAL { ?UserAcc sioc:name ?userName }
}
```

Query Q_7 is a construct query (line 2). Its result is a new RDF graph containing all tweets from the input stream that mention a musical artist (line 37) and some aggregated statistics: the sentiment score and like and share counts associated with each musical artist.

We executed query Q_7 with DSCEP using a network consisting of a single stream generator and a single *RSP:BasicProcessor Operator* (Figure 5.3). We measured the impact of varying the number of RSP engines ($\text{RSP}_{\texttt{number}}$) while keeping the window size ($\text{WIN}_{\texttt{size}}$) fixed and the impact of varying the window size ($\text{WIN}_{\texttt{size}}$) while keeping the number of RSP engines ($\text{RSP}_{\texttt{number}}$) fixed. Since we already evaluate the impact of the KB on the query processing, for this experiment, we did not vary the KB size (KB_{total}) and the query search-space (KB_{used}).

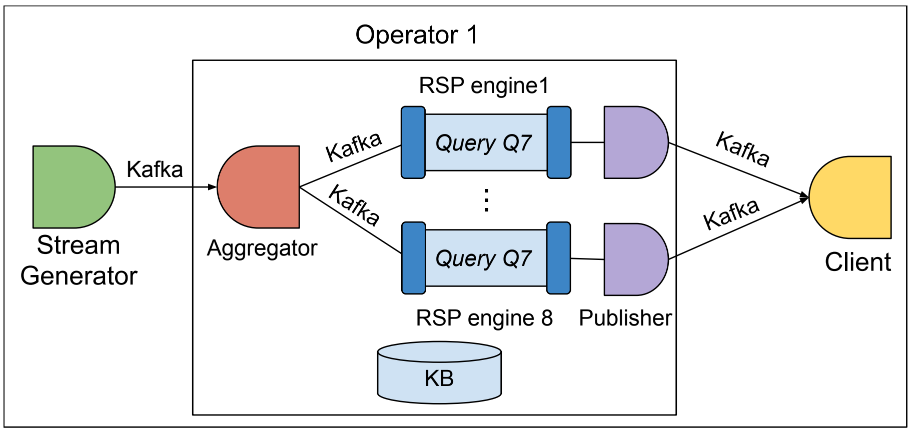

Figure 5.3: The parallelization of Q_7 using one operator with multiple RSP engines.

The results for this experiment are shown in Figure 5.4, which has four line graphs. Each graph represents a different test with a different operator setting. Each operator has a different number of RSP engines ($\text{RSP}_{\texttt{number}}$). First, we executed query Q_7 without intra-parallelism by using only one RSP engine (Operator 1 with $\text{RSP}_{\texttt{number}}$=1) and compared it with Operator 2, which has $\text{RSP}_{\texttt{number}}$=2. The processing time per window was, on average, 24.57s for Operator 1 with $\text{WIN}_{\texttt{size}}$=500, and 26.06s for Operator 2 for the same $\text{WIN}_{\texttt{size}}$. It means that the Operator 1 processes two windows in 49.14s (two times

24.57s), while the Operator 2 processes two windows (one in each RSP engine in parallel) in 26.06s. To be precise, Operator 2 has a processing time per window 6% higher than the processing time per window of Operator 1, but Operator 2 can process two windows in parallel. Thus, approximately, Operator 2 would take almost half of the time that Operator 1 takes to processes the same input. When increasing the window size, the processing time difference also increases between Operator 1 and Operator 2. With WIN$_{\texttt{size}}$=1000, the processing time per window of Operator 2 is 8.9% higher, with WIN$_{\texttt{size}}$=2000 is 10.4% higher and with WIN$_{\texttt{size}}$=5000 the processing time per window of Operator 2 is 9.8% higher. The processing time per window difference between operators was all inside the range of 6% to 11%. Also, when we increase the window size, the processing time difference per window between Operator 1 and Operator 2 increases and tends to stabilize around 10%.

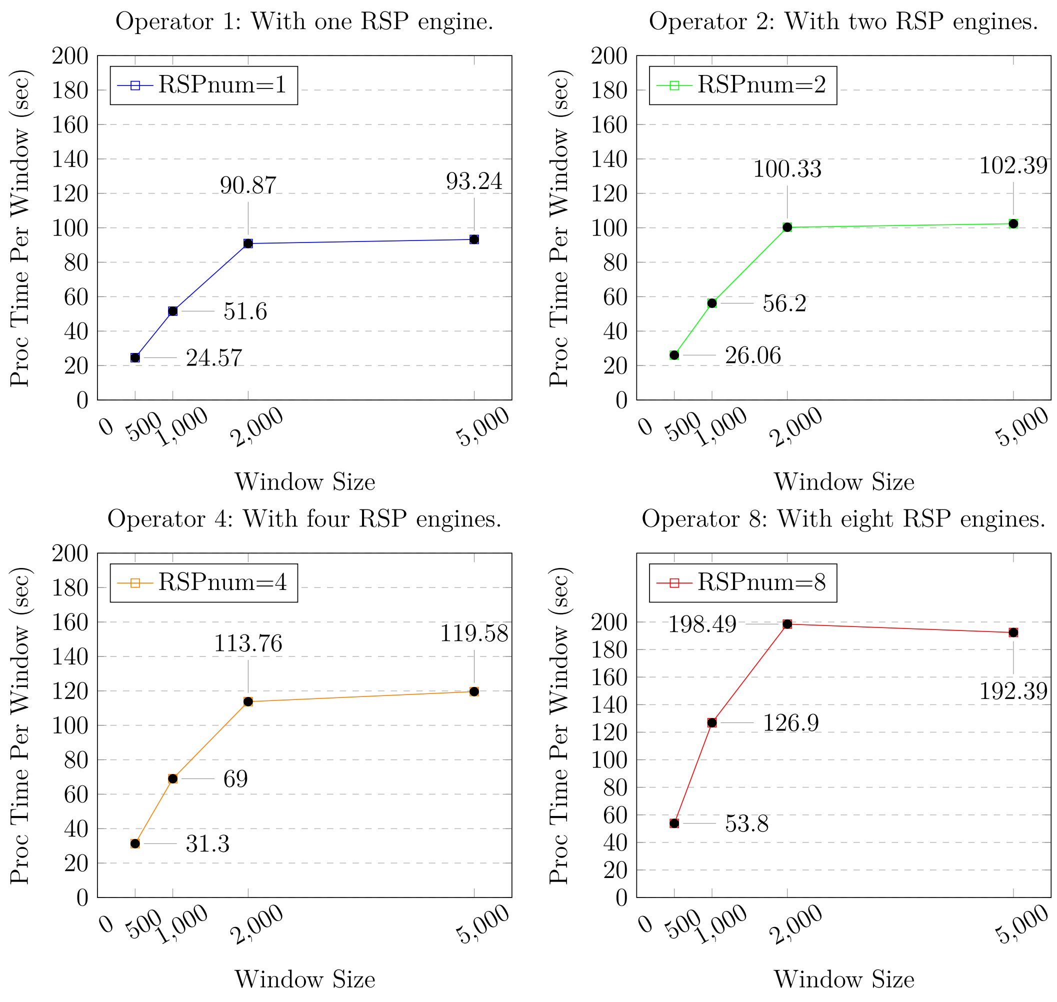

Figure 5.4: Average processing time per window with varying RSP$_{\texttt{number}}$and WIN$_{\texttt{size}}$for Q_7. All processing times (for each WIN$_{\texttt{size}}$) are an average of 4 runs with 1000 windows each.

When increasing RSP$_{\texttt{number}}$to 4 (with Operator 4), four windows with WIN$_{\texttt{size}}$=500 are processed in parallel in 31.3s, one on each RSP engine of

		Number of RSP engines							
	$\text{WIN}_{\texttt{size}}$	RSP engine 1	RSP engine 2	RSP engine 3	RSP engine 4	RSP engine 5	RSP engine 6	RSP engine 7	RSP engine 8
Operator 2	500	52.8%	47.2%						
Operator 4	500	26%	24.5%	25%	24.5%				
Operator 8	500	18%	16.9%	11.2%	7.9%	11.2%	13.5%	10.1%	11.2%
Operator 2	1000	52%	48%						
Operator 4	1000	26%	24%	26%	24%				
Operator 8	1000	12%	18%	12%	10%	12%	12%	14%	10%
Operator 2	2000	52.1%	47.9%						
Operator 4	2000	26.2%	25.1%	24.9%	23.8%				
Operator 8	2000	14.6%	10.4%	10.4%	8.3%	8.3%	14.6%	16.7%	16.7%
Operator 2	5000	50%	50%						
Operator 4	5000	27.2%	22.8%	22.8%	27.2%				
Operator 8	5000	8.3%	10.4%	6.3%	12.5%	18.8%	12.5%	10.4%	20.8%

Table 5.2: Window distribution among RSP engines of each operator (Percentage).

Operator 4. Operator 4, with $\text{WIN}_{\texttt{size}}$=500, has a processing time per window of 27.4% higher than the processing time per window of Operator 1. With $\text{WIN}_{\texttt{size}}$=1000 the processing time per window of Operator 4 is 33.7% higher, with $\text{WIN}_{\texttt{size}}$=2000 is 25.2% higher and with $\text{WIN}_{\texttt{size}}$=5000 is 28.2% higher. In this case, the processing time per window difference between operators 1 and 2 were all inside the range of 25% and 33%.

Different behavior can be observed when comparing the processing time per window of Operator 8 with Operator 1. Operator 8 takes 119% more time to process a window when compared to Operator 1, but also Operator 8 can process up to 8 windows in parallel. In seconds, Operator 1 takes an average of 24,57s to process a single window, and Operator 8 takes an average of 53.8s. The processing time of a single window increases much more on Operator 8 when compared to Operators 2 and 4. That is because the machine used for the test has only four cores, making it impossible to execute eight windows simultaneously. Since each core can execute one process per time, the Operator 4 (with $\text{RSP}_{\texttt{number}}$=4) would be the limit for this machine to guarantee that it is possible to execute four windows simultaneously.

The number of CPU cores impact on each operator can also be observed in Table 5.2, where Operator 8 tends to have a more significant difference on window distribution among its RSP engines compared to other operators. Operator 2 and Operator 4 show that even when increasing the $\text{WIN}_{\texttt{size}}$, the window distribution percentage was maintained approximately equally divided among each RSP engine. If there is at least one CPU core per RSP engine, the window distribution tends to be equally divided among RSP engines.

This third experiment demonstrates that intra-operator parallelism can reduce overall processing time. It also shows that the number of RDF processors executed in parallel on the same machine should be compatible with the number of CPU cores to avoid increasing the processing time of a single window and to get a more equally divided window distribution among RSP engines.

Note that because DSCEP's aggregator and publisher use Kafka to send and receive windows, it also enables that each RSP engine of a single operator can be executed on a different machine (as illustrated by Figure 5.5). This is a consequence of the capability that Kafka has to distribute and coordinate data streams.

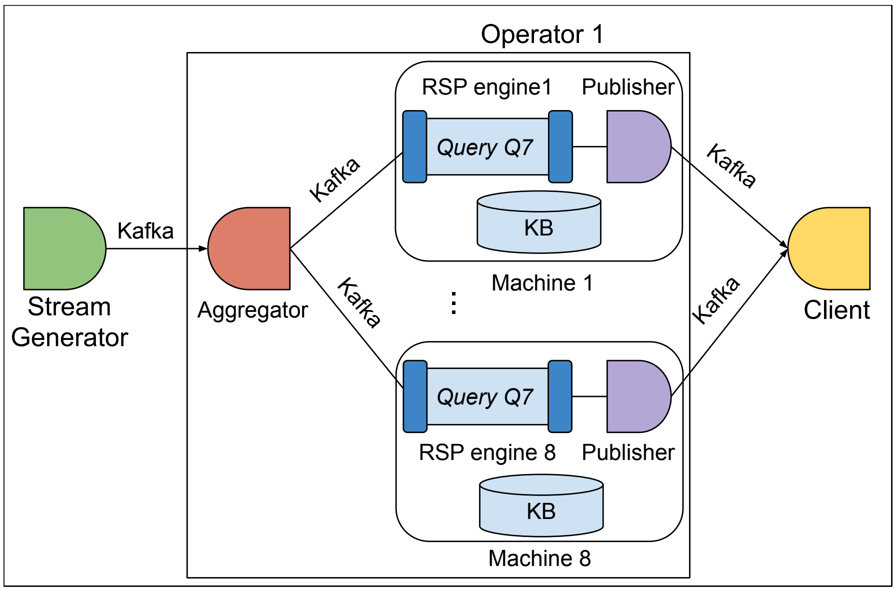

Figure 5.5: The parallelization of Q_7 using one operator with multiple RSP engines, each in a different machine.

To assign a number of RSP engines equal to the machine's number of CPU cores, allow the use of more RSP engines on a single operator without increasing the processing time per window.

6
Discussion and Limitations

This chapter discusses limitations and difficulties regarding dividing a query into smaller subqueries and challenges on RDF knowledge base partitioning. It is worth mentioning that a deep discussion and theoretic analysis of query partitioning, query processing, and KB partitioning are beyond the scope of this thesis. In the following sections, we give an overall idea and a brief discussion about these topics.

6.1
Remarks on dividing one query into multiple smaller queries

Query processing in an area in which one of the study subjects is how to decrease the processing time of a query. One of the ways of speeding up a query is to break the query into smaller subqueries, execute them in parallel, and then join its results. Before describing the challenges of breaking a query into multiple subqueries, we give a short introduction to the SPARQL query language and refresh some of the basic characteristics of the RDF language explained in chapter 2.

In the RDF language, all data are represented as a triple (s,p,o) consisting of a subject *s*, predicate *p*, and object *o*. These triples can be interlinked in a way that the object *o* of a triple can be the subject *s* of another triple. Thus, an RDF dataset can be considered as representing a directed graph, with entities (i.e., subjects and objects) as nodes and relationships (i,e. predicates) as directed edges. The core syntax of SPARQL is a conjunctive set of triple patterns called *basic graph pattern* [41]. A triple pattern is similar to an RDF triple, except that any component in the triple pattern can be a variable. We call a *basic graph pattern* a subgraph that a user wants to match against the RDF data. Thus, SPARQL query processing is essentially a subgraph matching problem with one or more basic triple patterns to match.

For example, let us analyze the following SPARQL query:

Listing 6.1: Example SPARQL query

```
REGISTER QUERY ExampleQuery1 AS
CONSTRUCT {
        ?temperatureSensor ex:hasSensorID ?tSensorID .
4         ?humiditySensor ex:hasSensorID ?uSensorID .
5         ex:Colocation ex:isColocated ?locTimes }
6 FROM STREAM <http://ex.org/Sensors> [RANGE TRIPLES 1000]
7 FROM <file:///Sensors-KB.rdf>
8 WHERE  {
9         ?temperatureSensor ex:hasSensorID ?tSensorID .
10        ?temperatureSensor ex:hasSensorType ?sensorType1 .
11        ?sensorType1 rdf:type ex:TemperatureSensor .
12        ?humiditySensor ex:hasSensorID ?uSensorID .
13        ?humiditySensor ex:hasSensorType ?sensorType2 .
14        ?sensorType2 rdf:type ex:HumiditySensor .
15
16        SELECT (count(?loc1) as ?locTimes)
17        WHERE {
18        ?temperatureSensor ex:hasLocation ?loc1 .
19        ?humiditySensor ex:hasLocation ?loc2 .
20        FILTER (?loc1 = ?loc2)
21        } }
```

This query aims to return a graph with three triples, where the first triple contains the ID of the temperature sensor, the second triple contains the ID of the humidity sensor, and the third triple describes if both sensors are co-located. The variable `?locTimes` identifies if both sensors are co-located or not, its value can be either 0 (not co-located) or 1 (is co-located).

The query can be decomposed into three basic graph patterns, lines 9, 10, and 11 represent the first, lines 12, 13 and 14 represents the second, and the third is represented from line 16 to line 20. The first and second subgraph patterns are similar, and they both can be executed in parallel since the task to find all temperature sensors does not depend on the task to find all humidity sensors. Thus, these two basic graph patterns do not depend on each other. On the other hand, the third basic graph pattern can not be executed in parallel because it needs data retrieved by the other basic graph patterns.

Thus, one way to divide the query represented in listing 6.1 is to divide it into three separate queries, in this case, one for each basic subgraph pattern and to execute them following the topology illustrated in figure 6.1.

Figure 6.1 is just one example of dividing a simple query that contains only three basic graph patterns into three subqueries. But queries can become more complex with more dependencies between basic graph patterns making the partitioning of the query a difficult task. In the case of this thesis, the challenge is not to divide one query into multiple subqueries but to ensure that each subquery can execute in a different machine and, in some cases, by a different RDF stream processor engine.

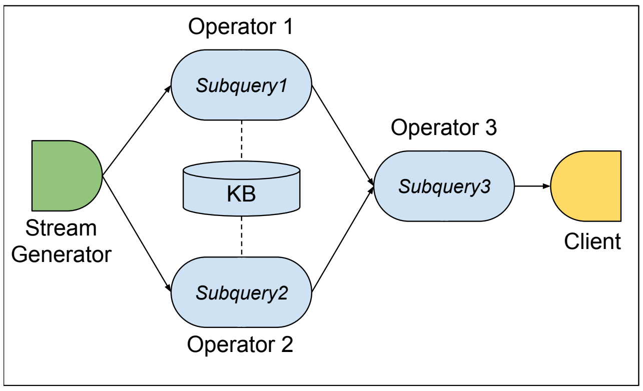

Figure 6.1: Topology to execute the subqueries.

6.2 Remarks on dividing a knowledge base

Knowledge bases are increasingly being used for representing data on the Web [42], usually for expert systems that work with one or possibly several domain areas. In this context, one of the challenges is scalable reasoning that can generate responsive results to complicated queries involving large datasets. In most cases, not all data in the knowledge base applies to and is required by every query formulated to be used in a specific use case. It is possible that by dividing the knowledge base, a decrease in execution time may be achieved. However, splitting a knowledge base into independent partitions is a complex problem because it is difficult to ensure complete and sound query reasoning on the partitioned data [43].

The complexity of partitioning a knowledge base into multiple parts is directly related to its level of expressiveness and formality. The more expressive a KB is, the inter dependency among its elements is higher [44].

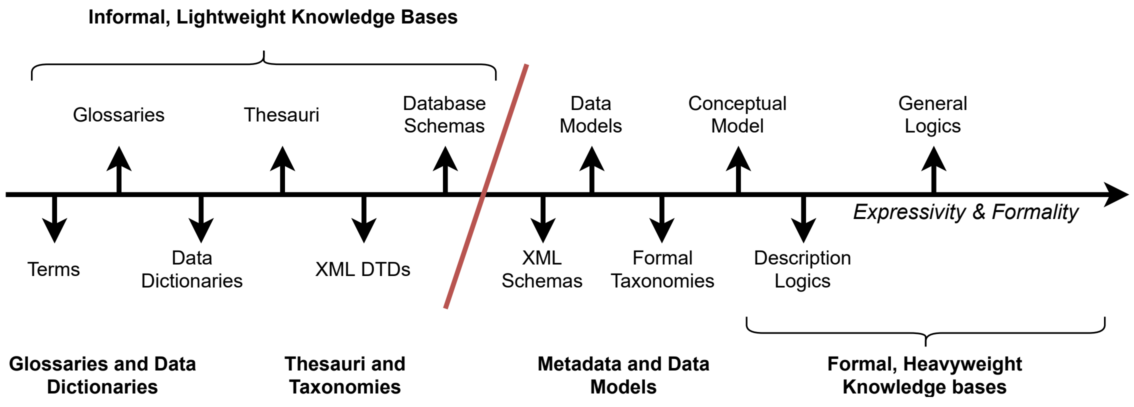

Figure 6.2: Knowledge base expressiveness levels [1].

Figure 6.2 shows a general vision about the levels of expressiveness of a knowledge base. Glossaries, Data Dictionaries, Thesauri, and Taxonomies are the less expressive models and thus easier to partition when compared with formal and heavyweight knowledge bases. The more expressive models tend to have implicit data generated explicitly when the reasoning is executed. Thus, the partitioning process can compromise the query result since reasoning with only a part of the knowledge base can result in different outcomes compared to reasoning with the complete KB.

The other characteristic that is important to have in mind when partitioning a knowledge base is to know all the queries that will be executed. If there is a way to know beforehand all queries that will be executed on the system, the KB partitioning process can be guided to support these specific queries. By partitioning a KB for only one query, the designer does not have to guarantee that for every other query, all results will be the same as when the query is executed on the complete KB [45].

Consider the example of listing 6.1 of the previous subsection, where we have three subqueries, and two of them need to access the knowledge base. One of the queries is interested in all temperature sensors, and the other is interested in all humidity sensors. Also, consider the RDF graph model used to represent all sensor instances, represented in figure 6.3.

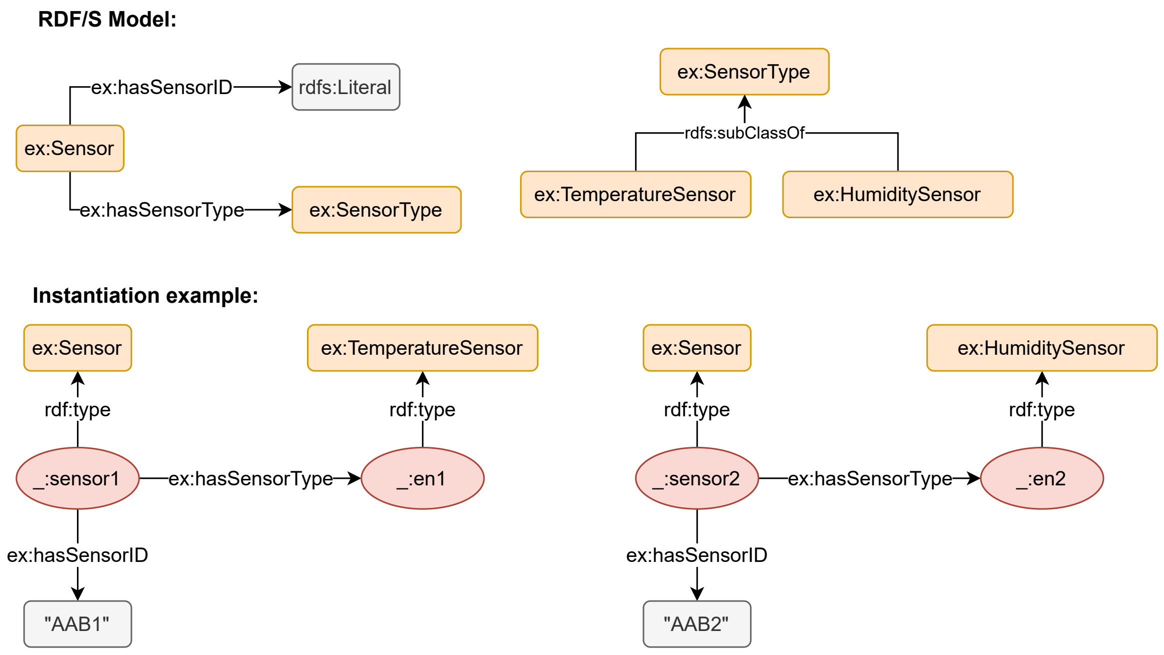

Figure 6.3: RDF graph model of the sensor data.

This knowledge base is written using RDF, and there is no rule or underlying model that connects temperature sensors with humidity sensors. For queries one and two, it is possible to divide the KB into two parts, one containing all instances of temperature sensors and the other containing all instances of humidity sensors.

The task of partitioning a KB can become more complex depending on its expressiveness level. To enable such partitioning for knowledge bases that are not lightweight is yet an open research problem. To give an example, in Cafezeiro *at al* work [46], it is proposed an algebra of contextualized ontologies that needs to be applied since the creation of the knowledge base to make possible it is partitioning. The theory created by Cafezeiro ensures that if the knowledge base is created following a set of rules that defines a contextualized ontology, the partitioning of the knowledge base will be possible. However, all partitions of the KB are pre-defined since the KB is first created, making this method unsuitable to be applied on already existent KBs.

6.3 Conclusion

Both query and knowledge base partitioning are hard problems, and there is no simple solution to partition every query or every KB. In DSCEP infrastructure, the operator is a shared-nothing architecture, and consequently, KB partitioning is one fundamental strategy to speed up the computation of the query and increase scalability. DSCEP operators handle RDF data, both on stream and the KB, and several distributed RDF processing systems have been introduced where the storage and query processing is managed on multiple nodes [47].

The use of distributed RDF storage and query systems is possible since each DSCEP operator has its own RDF stream processor and storage. Distributed RDF storage systems do not guarantee that each partition of the KB is independent of the other. It might incur significant intermediate data shuffling when answering complex SPARQL queries that span multiple different partitions. However, distributed RDF processing systems are characterized by larger memory sizes and higher processing capacity.

Within the data streaming environment, we believe that the most natural way to partition the data is by dividing the windows among multiple nodes to be processed in parallel. We are aware that partitioning the data stream into windows only solves the problem of local reasoning. To reason with historical data is a more complex problem that requires the persistence of the data on the stream. Finally, the knowledge base size can still be a bottleneck since every window will have to be processed with the KB data.

7
Related Work

The following characteristic was listed to analyze and compare each related work with DSCEP:

- Intra-query parallelism: This can be done by instantiating multiple RSP engines running the same query into one Operator; the windows of the same input RDF stream are divided among the RSP engines.
- Inter-query parallelism: Different queries can be executed in parallel by different operators, each running its set of RSP engines.
- Integrates existing RSPs: The infrastructure allows the use of different RSP engines, and processing can be done in a compositional manner (i.e., the output of an operator should be ready to be used as input for another operator in the network).
- Allows external KB access: The infrastructure allows its RSP engines to access local or external KBs.
- Does not restrict the input stream: The infrastructure does not apply any restriction to the input RDF stream.
- Complies with Semantic CEP: The infrastructure allows and offers features to each RDF stream processor to run as a Semantic CEP engine.

Semantic CEP often shows an increase in processing time due to the insertion of the KB within the stream processing; thus, the infrastructure should provide different parallelism methods to speed up query processing. The characteristic of integrating existing RSPs is not a requirement for Semantic CEP. However, it makes the infrastructure not depend on a specific RSP engine and allows the user to choose the RSP engine that is more suitable for his use case or a specific query. Finally, the three last characteristics listed above refer to the requirements that make an infrastructure to be compliant to Semantic CEP; these requirements are listed in section 2.4.

Several RSP engines have been proposed in the last decade, some focusing on the processing aspects of continuous RDF queries and others focusing on enhancing query expressiveness and reasoning capabilities [15, 16, 17, 18, 19, 20]. CQELS-cloud [48] was one of the first RSP engines to focus on scalability and was capable of executing on multiple machines. Its query analyzer uses Apache Storm to parallelize queries and HBase to store intermediate results and RDF static data. However, unlike DSCEP, CQELS-cloud focuses on parallelizing a single query execution and not on providing an infrastructure for coordinating the execution of multiple queries. Intra- and inter- query parallelism are not provided, since CQELS-cloud focus on parallelize the execution it self of the SPARQL language. Additionally, in CQELS-cloud, queries can only access local knowledge bases.

Calbimonte [2, 49] is one of the first proposals for distributed RDF stream processing using multiple RSPs. It focuses on connecting different RSP engines using W3C's Linked Data Notifications (LDN; [50]). LDN is a protocol that describes how servers (receivers) can have messages pushed to them by applications (senders) and how application (consumers) may retrieve those messages. Calbimonte's work extends the LDN protocol and uses it as the backbone for sending and receiving RDF stream elements, where RDF streams are interpreted as notifications by the applications sending and receiving them. Figure 7.1 shows an overview of Calbimonte's infrastructure, where each circle on the figure represents either a data stream generator, consumer, or processor. Calbimonte's infrastructure is implemented in Scala using the Akka HTTP library. The implementation focus on showing the feasibility of connecting different RSP engines, stream generators, and consumers.

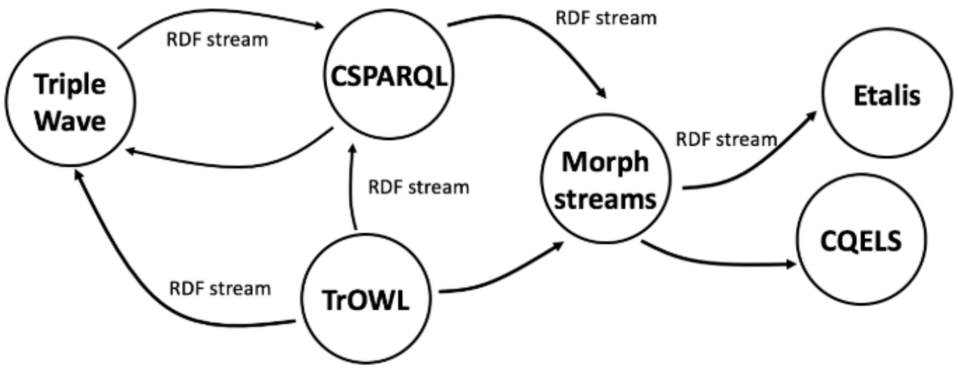

Figure 7.1: Calbimonte's infrastructure. [2]

One difference between Calbimonte's work and DSCEP is that DSCEP focuses on enabling the RSP engines to work as Semantic CEP operators. Calbimonte's proposal has restrictions that prevent this. For instance, in Calbimonte's proposal, RSP engines can only access local KBs. Also, while DSCEP permits intra- and inter-query parallelism, Calbimonte's only supports inter-query parallelism. DSCEP also has its window management and provides it to all RSP engines connected, enabling the use of RDF triple streams or RDF graphs streams.

Another difference is the communication paradigm used among the operators of the infrastructure. DSCEP uses a publish-subscribe communication which is asynchronous, and once it is deployed, all receivers will get their data as soon as they are published on the platform. While in the Calbimonte's infrastructure, one of their focus was to extend the LDN protocol, which is based on notifications exchange, to enable working with RDF data streams. One extension created by Calbimonte's work for the LDN was to create a new HTTP method for the consumer. Every time the consumer wants to retrieve data from a sender, it must first send a message to the sender to determine how long or how many events will be consumed. Another extension created is to enable the receiver to use the HTTP push method to check if there is new data and also to retrieve data directly from the sender.

Strider [3] is another infrastructure for distributed RDF stream processing, and figure 7.2 shows an overview of it. The left side of figure 7.2 shows the input data stream, which consists of transforming messages from IoT devices into an RDF serialization. Apache Kafka is used to handling the incoming messages from IoT devices and deliver these messages to the Spark Streaming layer. Strider's focus is to process input streams from previously connected IoT devices only and not to receive other data sources that were not previously added. The schema of the messages of all IoT devices connected to Strider is already known in advance, and they are previously mapped to ontology elements, facilitating optimization and stream processing.

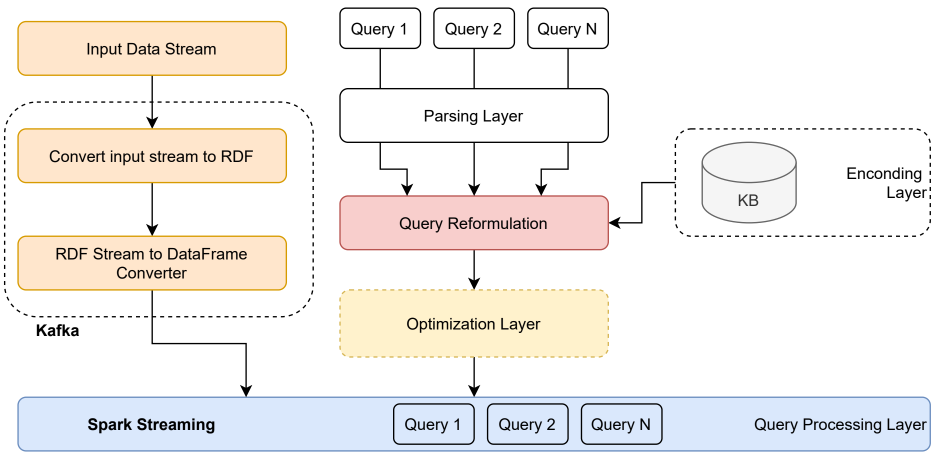

Figure 7.2: Strider's infrastructure. [3]

Strider uses its query language and translates its queries into Spark Streaming queries for enhanced parallelization. All knowledge bases used by all queries are previously encoded with the query so that every data necessary for query execution will be accessible locally. The encoding process of KBs restrict its expressiveness to a subset of RDFS with only `rdfs:subClassOf`, `rdfs:subProperty`, `rdfs:domain` and `rdfs:range` constructors. Strider queries cannot access external KBs and can not work with more expressive KBs (which uses more than the RDFS subset defined by Strider), limiting its use as a Semantic CEP solution.

BigSR [4] is an evolution of Strider which aims to improve some of its shortcomings. Figure 7.3 shows an overview of the system architecture. The *Data-Feed* (left part of the figure) maintains the same characteristics of Strider's data input stream ingestion, where data streams always came from IoT sensors and can be converted to RDF streams. The *Computing core* (center part of figure 7.3) contains the main modifications when compared to Strider. Instead of defining multiple queries, in BigSR, the user writes just one query in the form of a sequential code and can specify a set of rules to be applied to multiple sensors.

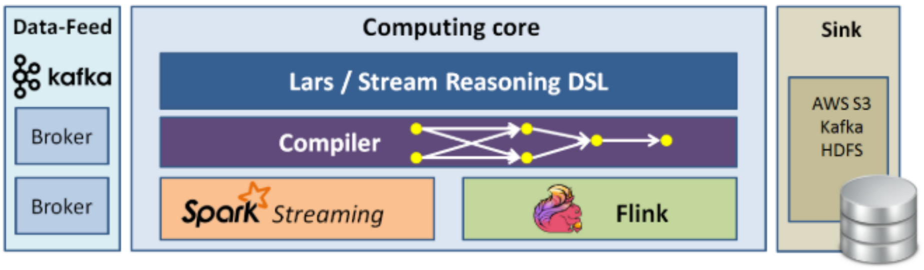

Figure 7.3: BigSR's infrastructure. [4]

BigSR enables recursive queries and adds supports for logical axioms and rules by using the domain-specific language used to write the query. Once the query is defined, the compiler on BigSR is responsible for making the query plan and parallelizing its execution. Consequentially, BigSR focuses on parallelizing the execution and automatically generating the query plan and not on providing an infrastructure for coordinating the user's execution of multiple different queries. BigSR still does not offer access to external background knowledge on query level.

Table 7.1 gives a comparison of the related work discussed above and DSCEP. From the table, it is clear that the primary differentiator of DSCEP is that it was designed and built with Semantic CEP in mind. For the sake of efficiency (and lower latency), recent solutions for distributed RDF stream processing tend to restrict external KB access. Although they gain in efficiency, they lose in expressivity and reasoning capability, which are critical features of RDF and Semantic Web technologies in general. Recent solutions also restrict the input RDF stream to optimize its processing. By restricting the input stream, it is possible to map the data stream schema in advance and improve efficiency. The DSCEP design philosophy goes in the opposite direction: we propose to reuse existing RSP solutions and to allow unrestricted access to external KBs; efficiency should be pursued through intra- and inter- query parallelization, window processing parallelization and by reducing their search space (by dividing the KBs).

Table 7.1: Related work vs DSCEP.

	CQELS-cloud [48]	Calbimonte's [2]	Strider [3]	BigSR [4]	DSCEP
Query intra-parallelism	−	−	+	+	+
Query inter-parallelism	−	+	+	+	+
Integrates existing RSPs	−	+	−	−	+
Allows external KB access	−	−	−	−	+
Does not restrict the input stream	+	+	−	−	+
Complies with Semantic CEP	−	−	−	−	+

8 Conclusion

As far as we know, DSCEP is one of the first distributed infrastructures focused on Semantic CEP, which enables reasoning with data both on the stream and on external or local KBs (Research question 1 - RQ1). DSCEP infrastructure enables the construction of distributed Semantic CEP operator networks using existing RSP engines by providing features to approximate current RSP solutions to the processing model of SCEP engines (Research questions 2 and 3 - RQ2 and RQ3). For example, by offering window management capabilities, support to multiple streams, support for streams of RDF-graphs, and support for processing streams in a compositional manner. Additionally, it enables newly created RDF stream processors to work with already existing ones.

We show how DSCEP distributed RDF stream processing can speed up monolithic SPARQL queries by breaking them into parallel subqueries (Research question 5 - RQ5). We did experiments that showed a considerable reduction in query processing time when such parallelization is applied.

We also discussed the possibility of partitioning large knowledge bases among the various subqueries so that not only the search space but the total KB side considered by each query is reduced (Research question 4 - RQ4). Our experiments indicate that KB partition can considerably impact the overall processing time of the system. The mere presence of unrelated triples (not part of the query search-space) can significantly slow down the query.

DSCEP provides an infrastructure that enables the test and evaluation of different strategies for query and KB partitioning. It is also possible to test and evaluate with DSCEP how operator and KB placement can affect the overall system performance.

Additionally, we showed that it is possible to reduce the processing time by parallelizing the windows of the same input stream into multiple nodes containing the same query. Our experiments show that by dividing the windows of a single stream into two parallel processing nodes, it is possible to reduce the query processing time by almost half. In the stream processing area, the partitioning of the stream into windows is essential. Consequently, speeding up a continuous query by parallelizing the processing of its windows is the most natural way of partitioning.

These are experiments that focus on specific characteristics to help us understand the impact of each part of the system on the overall performance. The system's final performance in the real world will depend on a set of additional variables, like CPU and memory capacity, latency among nodes, etc.

Although we used C-SPARQL and our basic implementation of stream processor as RSP engine in the experiments, DSCEP provides features that ease the integration of other RSP engines or any other RDF processing system. These features include stream aggregation and splitting, window management, RDF triple and RDF graph streams support, and inter-query and intra-query parallelism support.

We are aware that this is only a first and initial step towards real-time reasoning over data streams using background knowledge bases, and that much more theoretical and practical research is required to show its feasibility for large-scale and distributed applications. For instance, more work is still needed in SCEP engines because the scalability of current SCEP engines when processing data from the stream against a knowledge base is still an issue. This scalability problem may be mitigated by proposing new RDF processing algorithms and new distributed infrastructures.

Also, one possible line of research is to investigate how to automate the process of partitioning and distributing KB among the various operators. Supposing that the KB is flat and lightweight and since queries are often static, DSCEP could identify the part of the KB used by each operator and partition the KB accordingly.

The quality of service (QoS) is also an interserting topic to study. For the developer and the client to better understand the KB and query partitioning, it would be interesting to research and define the quality of service (QoS) parameters. The client could determine a set of QoS conditions that can facilitate or restrict the KB/query partitioning depending on the QoS chosen. For example, if the client does not want to apply reasoning, the KB partitioning problem becomes less complex. Some clients just want to use RDF as a format to integrate different data sources.

Another possible future work is to add to DSCEP the ability to determine and change operator and database placement on the fly. During runtime, DSCEP could monitor different infrastructure parameters and test different placements for operators and databases to optimize for things like overall processing time, latency, etc.

Bibliography

[1] PEASE, A.. **Ontology: A Practical Guide**. Articulate Software Press, 2011.

[2] CALBIMONTE, J.-P.. **Linked data notifications for rdf streams**. In: Proc. 2017 WSP and WOMoCoE (co-located ISWC), 2017.

[3] REN, X.; CURÉ, O.; KE, L.; LHEZ, J.; BELABBESS, B.; RANDRIAMALALA, T.; ZHENG, Y.; KEPEKLIAN, G.. **Strider: An adaptive, inference-enabled distributed RDF stream processing engine**. Proc. VLDB Endow., 10(12), 2017.

[4] REN, X.; CURÉ, O.; NAACKE, H.; XIAO, G.. **BigSR: Real-time expressive RDF stream reasoning on modern Big Data platforms**. In: Proc. 2018 IEEE Big Data. IEEE, 2018.

[5] LUCKHAM, D. C.. **The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems**. Addison-Wesley, 2001.

[6] DAYARATHNA, M.; PERERA, S.. **Recent advancements in event processing**. ACM Comput. Surv., 51(2), 2018.

[7] AGRAWAL, J.; DIAO, Y.; GYLLSTROM, D.; IMMERMAN, N.. **Efficient pattern matching over event streams**. In: Proc. 2008 SIGMOD. ACM, 2008.

[8] TEYMOURIAN, K.; PASCHKE, A.. **Towards semantic event processing**. In: Proc. 2009 DEBS. ACM, 2009.

[9] SCHAAF, M.; GRIVAS, S. G.; ACKERMANN, D.; DIEKMANN, A.; KOSCHEL, A.; ASTROVA, I.. **Semantic complex event processing**. In: Proc. 2012 WSEAS. WSEAS, 2012.

[10] TEYMOURIAN, K.; PASCHKE, A.. **Semantic enrichment of event stream for semantic situation awareness**. In: Semantic Web: Implications for Technologies and Business Practices. Springer, 2016.

[11] KESKISÄRKKÄ, R.. **Towards semantically enabled complex event processing**. Technical Report 1782, Linköping University, 2017.

[12] TEYMOURIAN, K.; ROHDE, M.; PASCHKE, A.. **Fusion of background knowledge and streams of events**. In: Proc. 2012 DEBS. ACM, 2012.

[13] SEQUEDA, J. F.; CORCHO, O.. **Linked data stream: A position paper**. In: Proc. 2009 SSN09 (co-located ISWC). CEUR-WS.org, 2009.

[14] **The W3C RDF Stream Processing Community Group**. https://www.w3.org/community/rsp/. Accessed May 2021.

[15] BOLLES, A.; GRAWUNDER, M.; JACOBI, J.. **Streaming SPARQL: Extending SPARQL to process data streams**. In: The Semantic Web: Research and Applications. Springer, 2008.

[16] BARBIERI, D. F.; BRAGA, D.; CERI, S.; VALLE, E. D.; GROSSNIKLAUS, M.. **Querying RDF streams with C-SPARQL**. SIGMOD Record, 39(1), 2010.

[17] CALBIMONTE, J.-P.; CORCHO, O.; GRAY, A. J. G.. **Enabling ontology-based access to streaming data sources**. In: Proc. 2010 ISWC. Springer, 2010.

[18] ANICIC, D.; FODOR, P.; RUDOLPH, S.; STOJANOVIC, N.. **Ep-sparql: A unified language for event processing and stream reasoning**. In: Proc. 2011 WWW. ACM, 2011.

[19] LE-PHUOC, D.; DAO-TRAN, M.; PARREIRA, J. X.; HAUSWIRTH, M.. **A native and adaptive approach for unified processing of linked streams and linked data**. In: Proc. 2011 ISWC. Springer, 2011.

[20] KIETZ, J.-U.; SCHARRENBACH, T.; FISCHER, L.; BERNSTEIN, A.; NGUYEN, K.. **TEF-SPARQL: The DDIS query-language for time annotated event and fact triple-streams**. Technical report, University of Zurich, Department of Informatics, 2013.

[21] GILLANI, S.; ZIMMERMANN, A.; PICARD, G.; LAFOREST, F.. **A query language for semantic complex event processing: Syntax, semantics and implementation**. Semantic Web, 10, 2018.

[22] REIS, R. D.; ENDLER, M.; DE ALMEIDA, V. P.; HAEUSLER, E. H.. **A soft real-time stream reasoning service for the internet of things**. In: 2019 IEEE 13th International Conference on Semantic Computing (ICSC), p. 166–169, 2019.

[23] DE ALMEIDA, V. P.; BHOWMIK, S.; LIMA, G.; ENDLER, M.; ROTHERMEL, K.. **Dscep: An infrastructure for decentralized semantic complex event processing**. In: 2020 IEEE International Conference on Big Data (Big Data), p. 391–398, 2020.

[24] ENDLER, M.; E SILVA, F. S.. **Past, present and future of the contextnet iomt middleware**. Open Journal of Internet Of Things, 4(1), 2018. Special Issue: Proc. 2018 VLIoT (co-located VLDB).

[25] KOLDEHOFE, B.; MAYER, R.; RAMACHANDRAN, U.; ROTHERMEL, K.; VÖLZ, M.. **Rollback-recovery without checkpoints in distributed event processing systems**. In: Proc. 2013 DEBS. ACM, 2013.

[26] WOOD, D.; LANTHALER, M.; CYGANIAK, R.. **RDF 1.1 concepts and abstract syntax**. Recommendation, W3C, 2014.

[27] CAROTHERS, G.; PRUDHOMMEAUX, E.. **RDF 1.1 Turtle**. Recommendation, W3C, 2014.

[28] HAYES, P.; PATEL-SCHNEIDER, P.. **RDF 1.1 semantics**. W3C recommendation, W3C, 2014.

[29] BRICKLEY, D.; GUHA, R.. **RDF schema 1.1**. W3C recommendation, W3C, 2014.

[30] W3C-OWL-WG-2012. **OWL 2 web ontology language document overview (second edition)**. W3C recommendation, W3C, 2012.

[31] ANTONIOU, G.; VAN HARMELEN, F.. **Web ontology language: OWL**. In: Handbook on Ontologies. Springer, 2009.

[32] ZOU, L.; ÖZSU, M. T.. **Graph-Based RDF Data Management**. Data Science and Engineering, 2(1):56–70, 2017.

[33] DELL'AGLIO, D.; VALLE, E. D.; VAN HARMELEN, F.; BERNSTEIN, A.. **Stream reasoning: A survey and outlook**. Data Sci., 1, 2017.

[34] KESKISÄRKKÄ, R.; BLOMQVIST, E.. **Semantic complex event processing for social media monitoring: A survey**. In: Proc. 2013 SMILE (co-located ESWC). CEUR-WS.org, 2013.

[35] TOMMASINI, R.; BONTE, P.. **Web stream processing with rsp4j**. In: Proceedings of the 15th ACM International Conference on Distributed and Event-Based Systems, DEBS '21, p. 164–167, New York, NY, USA, 2021. Association for Computing Machinery.

[36] EUGSTER, P. T.; FELBER, P. A.; GUERRAOUI, R.; KERMARREC, A.-M.. **The many faces of publish/subscribe**. ACM Comput. Surv., 35(2), 2003.

[37] FAFALIOS, P.; IOSIFIDIS, V.; NTOUTSI, E.; DIETZE, S.. **TweetsKB: A public and large-scale RDF corpus of annotated tweets**. In: Proc. 2018 ESWC. Springer, 2018.

[38] DE OLIVEIRA AVELINO, J.; DE FARIA CORDEIRO, K.; CAVALCANTI, M. C.. **An rdf based approach for integrating data at different levels of abstraction**. In: Proceedings of the Brazilian Symposium on Multimedia and the Web, p. 81–88, New York, NY, USA, 2020. Association for Computing Machinery.

[39] PAGE, K. R.; LEWIS, D.; WEIGL, D. M.. **Meld: A linked data framework for multimedia access to music digital libraries**. In: 2019 ACM/IEEE Joint Conference on Digital Libraries (JCDL), p. 434–435, 2019.

[40] XIONG, H.; PANDEY, G.; STEINBACH, M.; KUMAR, V.. **Enhancing data analysis with noise removal**. IEEE Transactions on Knowledge and Data Engineering, 18(3):304–319, 2006.

[41] QUILITZ, B.; LESER, U.. **Querying distributed RDF data sources with SPARQL**. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), 5021 LNCS:524–538, 2008.

[42] SAKR, S.; WYLOT, M.; MUTHARAJU, R.; LE PHUOC, D.; FUNDULAKI, I.. **Linked data: Storing, querying, and reasoning**. Springer International Publishing, March 2018.

[43] OREN, E.; KOTOULAS, S.; ANADIOTIS, G.; SIEBES, R.; TEN TEIJE, A.; VAN HARMELEN, F.. **Marvin: A platform for large-scale analysis of semantic web data**. In: Proc. of the WebSci09: Society On-Line, 2009.

[44] RAZ, T.; BOTTEN, N. A.. **The knowledge base partitioning problem: Mathematical formulation and heuristic clustering**. Data and Knowledge Engineering, 8(4):329–337, 1992.

[45] KOSSMANN, J.; PAPENBROCK, T.; NAUMANN, F.. **Data dependencies for query optimization: a survey**. VLDB Journal, 2021.

[46] CAFEZEIRO, I.; VITERBO, J.; RADEMAKER, A.; HAEUSLER, E. H.; ENDLER, M.. **Specifying ubiquitous systems through the algebra of contextualized ontologies**. Knowledge Engineering Review, 29(2):171–185, 2014.

[47] SAKR, S.; WYLOT, M.; MUTHARAJU, R.; PHUOC, D.; FUNDULAKI, I.. **Distributed RDF Query Processing**, p. 51–83. Springer International Publishing, 2018.

[48] LE-PHUOC, D.; QUOC, H. N. M.; VAN, C. L.; HAUSWIRTH, M.. **Elastic and scalable processing of linked stream data in the cloud**. In: Proc. 2013 ISWC. Springer, 2013.

[49] CALBIMONTE, J.-P.. **Decentralized messaging for rdf stream processing on the web**. Semantic Web – Interoperability, Usability, Applicability an IOS Press Journal, 2017.

[50] CAPADISLI, S.; GUY, A.. **Linked data notifications**. W3C recommendation, W3C, 2017.

A
Appendix

A.1
Queries

Query Q used on experiment 1 of chapter 5, subsection 5.5 is shown below:

```
REGISTER STREAM TweetStream AS
PREFIX ex: <http://example.org/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX owl: <https://www.w3.org/2002/07/owl#>
PREFIX onto: <file:///Users/vitor/git-repository/KAFKA/scep-
    operator/examples/>
PREFIX onyx: <http://www.gsi.dit.upm.es/ontologies/onyx/ns#>
PREFIX dc: <http://purl.org/dc/terms/>
PREFIX dbc: <http://dbpedia.org/page/Category:>
PREFIX dbo: <http://dbpedia.org/ontology/>
PREFIX dbp: <http://dbpedia.org/property/>
PREFIX dbr: <http://dbpedia.org/resource/>
PREFIX sioc: <http://rdfs.org/sioc/ns#>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX sioc_t: <http://rdfs.org/sioc/types#>
PREFIX schema: <http://schema.org/>
PREFIX nee: <http://www.ics.forth.gr/isl/oae/core#>
PREFIX wna: <http://www.gsi.dit.upm.es/ontologies/wnaffect/ns#>
# Get all tweets mentioning both musical artists and TV shows
# plus the aggregated sentiment scores, likes, and shares.
CONSTRUCT {
?tweet rdf:type sioc:Post .
?tweet sioc:id ?id .
?tweet dc:created ?datetime .
?tweet sioc:has_creator ?postCreator .
?tweet schema:interactionStatistic ?interactionSet .
?tweet schema:interactionStatistic ?interactionSet2 .
?interactionSet rdf:type schema:InteractionCounter .
?interactionSet schema:interactionType schema:LikeAction .
?interactionSet schema:userInteractionCount ?likeCount .
?interactionSet2 rdf:type schema:InteractionCounter .
?interactionSet2 schema:interactionType schema:ShareAction .
?interactionSet2 schema:userInteractionCount ?shareCount .
```

```
?artistURI ex:hasPositiveNumber ?countArtPositveTweets .
?artistURI ex:hasNegativeNumber ?countArtNegativeTweets .
?artistURI ex:hasLikeCount ?likesArt .
?artistURI ex:hasShareCount ?sharesArt .
?emotionSet onyx:hasEmotion ?negative .
?negative onyx:hasEmotionCategory wna:negative-emotion .
?negative onyx:hasEmotionIntensity ?negNum .
?tweet schema:mentions ?entity .
?entity nee:hasMatchedURI ?artistURI .
?artistURI dbo:genre ?genre .
?artistURI rdf:type dbo:MusicalArtist .
?entity ex:hasName ?name .
?tweet schema:mentions ?entity2 .
?entity2 nee:hasMatchedURI ?uri .
?entity2 ex:hasName ?nameEntity .
?uri rdf:type dbo:TelevisionShow .
?uri ex:hasPositiveNumber ?countTvPositveTweets .
?uri ex:hasNegativeNumber ?countTvNegativeTweets .
?uri ex:hasLikeCount ?likesTv .
?uri ex:hasShareCount ?sharesTv .
?tweet schema:mentions ?TagClass .
?TagClass rdf:type sioc_t:Tag .
?TagClass rdfs:label ?tag .
?tweet schema:mentions ?UserAcc .
?UserAcc rdf:type sioc:UserAccount .
?UserAcc sioc:name ?userName .
}
FROM STREAM <.../Tweets> [RANGE TRIPLES 1000]
FROM <.../KB.rdf>   # (Only for local access.)
WHERE {
# Q1: Get all tweets mentioning a musical artist.
# Q2: Get all tweets mentioning a TV show.
?artistURI rdf:type dbo:MusicalArtist .
?artistURI dbo:genre ?genre .
?tweet rdf:type sioc:Post .
?tweet sioc:id ?id .
?tweet dc:created ?datetime .
?tweet sioc:has_creator ?postCreator .
?tweet onyx:hasEmotionSet ?emotionSet .
?tweet schema:interactionStatistic ?interactionSet .
?tweet schema:interactionStatistic ?interactionSet2 .
?interactionSet rdf:type schema:InteractionCounter .
?interactionSet schema:interactionType schema:LikeAction .
?interactionSet schema:userInteractionCount ?likeCount .
?interactionSet2 rdf:type schema:InteractionCounter .
?interactionSet2 schema:interactionType schema:ShareAction .
?interactionSet2 schema:userInteractionCount ?shareCount .
```

```
?emotionSet onyx:hasEmotion ?positive .
?positive onyx:hasEmotionCategory wna:positive-emotion .
?positive onyx:hasEmotionIntensity ?posNum .
?emotionSet onyx:hasEmotion ?negative .
?negative onyx:hasEmotionCategory wna:negative-emotion .
?negative onyx:hasEmotionIntensity ?negNum .
?tweet schema:mentions ?entity .
OPTIONAL {?tweet schema:mentions ?entity2 .}
?entity nee:hasMatchedURI ?artistURI .
?entity nee:detectedAs ?name .
OPTIONAL {?entity2 nee:hasMatchedURI ?uri .    }
OPTIONAL {?entity2 nee:detectedAs ?nameEntity . }
OPTIONAL { ?uri rdf:type dbo:TelevisionShow . }
OPTIONAL { ?uri dbo:genre ?tvShowGenre . }
FILTER (?uri != ?artistURI)
OPTIONAL { ?tweet schema:mentions ?TagClass }
OPTIONAL { ?TagClass rdf:type sioc_t:Tag }
OPTIONAL { ?TagClass rdfs:label  ?tag }
OPTIONAL { ?tweet schema:mentions ?UserAcc }
OPTIONAL { ?UserAcc rdf:type sioc:UserAccount }
OPTIONAL { ?UserAcc sioc:name ?userName }

# Q3: Aggregate sentiment score of musical artists.
{
SELECT ?artistURI ?tweet ?otherURI ?genre (count(?posNum) as ?
    countArtPositveTweets)
WHERE
{
    ?tweet rdf:type sioc:Post .
    ?tweet onyx:hasEmotionSet ?emotionSet .
    ?tweet schema:mentions ?entity .
    ?tweet schema:mentions ?entityOther .
    ?entity nee:hasMatchedURI ?artistURI .
    ?artistURI rdf:type dbo:MusicalArtist .
    ?artistURI dbo:genre ?genre .
    ?emotionSet onyx:hasEmotion ?positive .
    ?positive onyx:hasEmotionCategory wna:positive-emotion .
    ?positive onyx:hasEmotionIntensity ?posNum .
    ?entityOther nee:hasMatchedURI ?otherURI .
    ?otherURI ex:type ex:isNotMusicalArtists .

    FILTER (!contains( str(?posNum), "0.0"))
}
GROUP BY ?artistURI ?tweet ?otherURI ?genre
}
{
```

```
SELECT ?artistURI2 ?tweet2 ?otherURI2 ?genre2 (count(?negNum)
   as ?countArtNegativeTweets)
WHERE
{
    ?tweet2 rdf:type sioc:Post .
    ?tweet2 onyx:hasEmotionSet ?emotionSet2 .
    ?tweet2 schema:mentions ?entity2 .
    ?tweet2 schema:mentions ?entityOther2 .
    ?entity2 nee:hasMatchedURI ?artistURI2 .
    ?artistURI2 rdf:type dbo:MusicalArtist .
    ?artistURI2 dbo:genre ?genre2 .
    ?emotionSet2 onyx:hasEmotion ?negative .
    ?negative onyx:hasEmotionCategory wna:negative-emotion .
    ?negative onyx:hasEmotionIntensity ?negNum .
    ?entityOther2 nee:hasMatchedURI ?otherURI2 .
    ?otherURI2 ex:type ex:isNotMusicalArtists .


    FILTER (!contains( str(?negNum), "0.0"))
}
GROUP BY ?artistURI2 ?tweet2 ?otherURI2 ?genre2
}

# Q4: Aggregate likes/shares of musical artists.
{
SELECT ?artistURI ?tweet ?genre (count(?likeCount) as ?likesArt
   )
WHERE
{
    ?tweet rdf:type sioc:Post .
    ?tweet schema:mentions ?entity .
    ?entity nee:hasMatchedURI ?artistURI .
    ?artistURI rdf:type dbo:MusicalArtist .
    ?artistURI dbo:genre ?genre .
    ?tweet schema:interactionStatistic ?interactionSet .
    ?interactionSet rdf:type schema:InteractionCounter .
    ?interactionSet schema:interactionType schema:LikeAction .
    ?interactionSet schema:userInteractionCount ?likeCount .
}
GROUP BY ?artistURI ?tweet ?genre
}
{
SELECT ?artistURI2 ?tweet2 ?genre2 (count(?shareCount) as ?
   sharesArt)
WHERE
{
    ?tweet2 rdf:type sioc:Post .
    ?tweet2 schema:mentions ?entity2 .
    ?entity2 nee:hasMatchedURI ?artistURI2 .
    ?artistURI2 rdf:type dbo:MusicalArtist .
    ?artistURI2 dbo:genre ?genre2 .
    ?tweet2 schema:interactionStatistic ?interactionSet2 .
    ?interactionSet2 rdf:type schema:InteractionCounter .
    ?interactionSet2 schema:interactionType schema:ShareAction
    .
    ?interactionSet2 schema:userInteractionCount ?shareCount .
}
GROUP BY ?artistURI2 ?tweet2 ?genre2
}

# Q5: Aggregate sentiment score of TV shows.
{
SELECT ?tvShowURI ?tweet ?otherURI ?genre (count(?posNum) as ?
    countTvPositveTweets)
WHERE
{
    ?tweet rdf:type sioc:Post .
    ?tweet onyx:hasEmotionSet ?emotionSet .
    ?tweet schema:mentions ?entity .
    ?tweet schema:mentions ?entityOther .
    ?entity nee:hasMatchedURI ?tvShowURI .
    ?tvShowURI rdf:type dbo:TelevisionShow .
    ?tvShowURI dbo:genre ?genre .
    ?emotionSet onyx:hasEmotion ?positive .
    ?positive onyx:hasEmotionCategory wna:positive-emotion .
    ?positive onyx:hasEmotionIntensity ?posNum .
    ?entityOther nee:hasMatchedURI ?otherURI .
    ?otherURI ex:type ex:TelevisionShow .

    FILTER (!contains( str(?posNum), "0.0"))
}
GROUP BY ?tvShowURI ?tweet ?otherURI ?genre
}
{
SELECT ?tvShowURI2 ?tweet2 ?otherURI2 ?genre2 (count(?negNum)
    as ?countTvNegativeTweets)
WHERE
{
    ?tweet2 rdf:type sioc:Post .
    ?tweet2 onyx:hasEmotionSet ?emotionSet2 .
    ?tweet2 schema:mentions ?entity2 .
    ?tweet2 schema:mentions ?entityOther2 .
    ?entity2 nee:hasMatchedURI ?tvShowURI2 .
    ?tvShowURI2 rdf:type dbo:TelevisionShow .
    ?tvShowURI2 dbo:genre ?genre2 .
    ?emotionSet2 onyx:hasEmotion ?negative .
    ?negative onyx:hasEmotionCategory wna:negative-emotion .
    ?negative onyx:hasEmotionIntensity ?negNum .
    ?entityOther2 nee:hasMatchedURI ?otherURI2 .
    ?otherURI2 ex:type ex:isNotTelevisionShow .


    FILTER (!contains( str(?negNum), "0.0"))
}
GROUP BY ?tvShowURI2 ?tweet2 ?otherURI2 ?genre2
}

# Q_6: Aggregate likes/shares of TV shows.
{
SELECT ?tvShowURI ?tweet ?genre (count(?likeCount) as ?likesTv)
WHERE
{
    ?tweet rdf:type sioc:Post .
    ?tweet schema:mentions ?entity .
    ?entity nee:hasMatchedURI ?tvShowURI .
    ?tvShowURI rdf:type dbo:TelevisionShow .
    ?tvShowURI dbo:genre ?genre .
    ?tweet schema:interactionStatistic ?interactionSet .
    ?interactionSet rdf:type schema:InteractionCounter .
    ?interactionSet schema:interactionType schema:LikeAction .
    ?interactionSet schema:userInteractionCount ?likeCount .
}
GROUP BY ?tvShowURI ?tweet ?genre
}
{
SELECT ?tvShowURI2 ?tweet2 ?genre2 (count(?shareCount) as ?
   sharesTv)
WHERE
{
    ?tweet2 rdf:type sioc:Post .
    ?tweet2 schema:mentions ?entity2 .
    ?entity2 nee:hasMatchedURI ?tvShowURI2 .
    ?tvShowURI2 rdf:type dbo:TelevisionShow .
    ?tvShowURI2 dbo:genre ?genre2 .
    ?tweet2 schema:interactionStatistic ?interactionSet2 .
    ?interactionSet2 rdf:type schema:InteractionCounter .
    ?interactionSet2 schema:interactionType schema:ShareAction
   .
    ?interactionSet2 schema:userInteractionCount ?shareCount .
}
GROUP BY ?tvShowURI2 ?tweet2 ?genre2
```

```
}
```